C

Caching for SQL-on-Hadoop

Gene Pang and Haoyuan Li
Alluxio Inc., San Mateo, CA, USA

Definitions

Caching for SQL-on-Hadoop are techniques and systems which store data to provide faster access to that data, for Structured Query Language (SQL) engines running on the Apache Hadoop ecosystem.

Overview

The Apache Hadoop software project (Apache Hadoop 2018) has grown in popularity for distributed computing and big data. The Hadoop stack is widely used for storing large amounts of data, and for large-scale, distributed, and fault-tolerant data processing of that data. The Hadoop ecosystem has been important for organizations to extract actionable insight from the large volumes of collected data, which is difficult or infeasible for traditional data processing methods.

The main storage system for Hadoop is the Hadoop Distributed File System (HDFS). It is a distributed storage system which provides fault-tolerant and scalable storage. The main data processing framework for Hadoop is MapReduce, which is based on the Google MapReduce project (Dean and Ghemawat 2008). MapReduce is a programming model and distributed batch processing framework for reliably processing large volumes of data, typically from HDFS. However, the distinct programming model for MapReduce can be a barrier for nondevelopers, such as data analysts or business intelligence (BI) tools.

Because of this barrier to entry, new tools and frameworks have emerged for the Hadoop ecosystem. Primarily, new Structured Query Language (SQL) frameworks have gained popularity for the Hadoop software stack. SQL is a domain-specific declarative language to describe retrieving and manipulating data, typically from relational database management systems (RDBMS). These SQL engines for Hadoop have been effective for empowering more users who are familiar with existing tools and the SQL language for managing data. These frameworks typically read data from HDFS, process the data in a distributed way, and return to the user the desired answer. Many SQL engines are available for Hadoop, which include Apache Hive (2018), Apache Spark SQL (2018), Apache Impala (2018), Presto (Facebook 2018), and Apache Drill (2018).

These SQL on Hadoop engines have combined the convenience of data query with SQL and the power of distributed processing of data with Hadoop. However, these engines are commonly not tightly integrated into the Hadoop ecosystem. There can be advantages of separating the computation engine from the storage system, such as cost effectiveness and operational flexibility.

S. Sakr, A. Y. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-77525-8

However, one potential weakness is that the performance of accessing data may decline. Since RDBMS typically have tight integration and control of the entire stack from the computation to the storage, performance may be faster and more predictable. For comparable experiences with traditional SQL processing in RDBMS, distributed SQL engines on Hadoop turn to caching to achieve the desired performance.

There are several different ways SQL engines on Hadoop can take advantage of caching to improve performance. The primary methods are as follows:

Internal caching within the SQL engine
Utilizing external storage systems for caching

SQL Engine Internal Caching

A common way a SQL engine uses caching is to implement its own internal caching. This provides the most control for each SQL engine as to what data to cache and how to represent the cached data. This internal cache is also most likely the fastest to access, since it is located closest to the computation being performed (same memory address space or local storage), in the desired format. However, the SQL engine internal caching may not be shareable between different users and queries, which is beneficial in a multi-tenant environment. Also, not all SQL processing frameworks have implemented an internal cache, which prevents using cached data for queries.

Apache Spark SQL

Apache Spark SQL is a SQL query engine built on top of Apache Spark, a distributed data processing framework. Spark SQL exposes a concept called a DataFrame, which represents a distributed collection of structured data, similar to a table in an RDBMS. Caching in Spark SQL utilizes the caching in the Spark computation engine. A user can choose to cache a DataFrame in Spark SQL by explicitly invoking a cache command (Spark SQL 2018). When the command is invoked, Spark SQL engine will cache the DataFrame internal to the Spark engine. There are several user-configurable ways the data can be cached, including MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, and others (Spark RDD 2018). An explicit command must be invoked to remove the DataFrame from the cache.

Apache Hive and LLAP

Apache Hive is a SQL data warehouse for Hadoop, which utilizes an existing distributed computation engine for the data processing. The options of computation engines for Hive to use are MapReduce, Spark, or Tez. Hive uses Live Long And Process (LLAP) (Apache Hive LLAP 2018) for caching data for SQL processing. Currently, only Tez will take advantage of the caching available via LLAP.

LLAP is a long-lived daemon which runs alongside Hadoop components in order to provide caching and data pre-fetching benefits. Since Tez takes advantage of LLAP, when Hive uses Tez as the computation engine, data access will go to the LLAP daemons, instead of the HDFS DataNodes. With this access pattern and architecture, LLAP daemons can cache data for Hive queries.

External Storage Systems for Caching

Another major technique for caching for SQL engines on Hadoop is to utilize an external storage system for caching data. Using a separate system can have several benefits. An external system may be able to manage the cache more effectively and provide additional cache-related features. Also, an external system can be deployed independently from the other components of the ecosystem, which can provide operational flexibility. Additionally, sharing cached data can be enabled or made simpler with a separate system for caching. By using a separate system for handling the caching of data, SQL engines can take advantage by reading the data from the cache, instead of the originating data source. This can greatly improve performance, especially when frequently accessed files are cached. Below

are some of the main systems which can provide caching in Hadoop ecosystems.

Alluxio

Alluxio (2018) is a memory speed virtual distributed storage system. Alluxio provides a unified namespace across multiple disparate storage systems. Alluxio enables users to "mount" any existing storage system like HDFS, Amazon S3, or a storage appliance, and presents a single, unified namespace encompassing all the data. When a mount is established, applications simply interact with the Alluxio namespace, and the data will be transparently accessed from the mounted storage and cached in Alluxio. Since there is no limit to how many mounts are possible, Alluxio enables accessing all data from a single namespace and interface, and allows queries to span multiple different data sources seamlessly. Figure 1 shows how Alluxio can mount multiple storage systems for multiple computation frameworks.

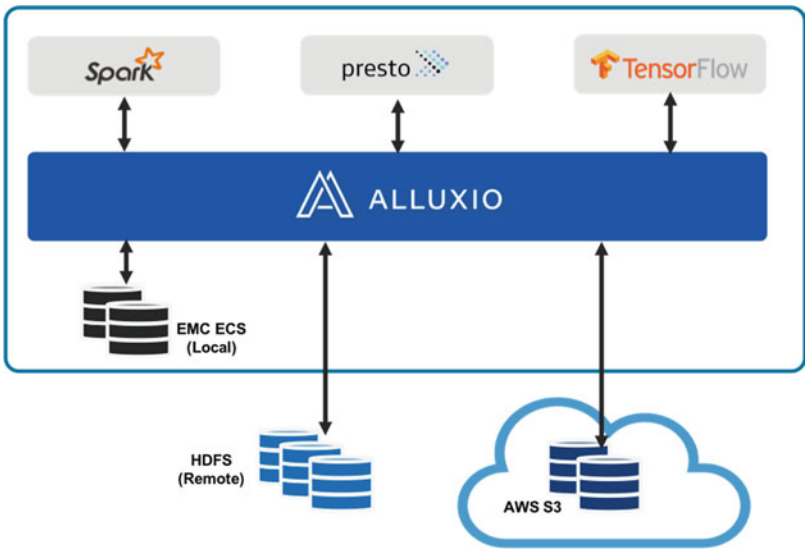
Alluxio enables caching for SQL engines by providing memory speed access to data via its memory centric architecture. Alluxio stores data on different tiers, which include memory, solid-state drive (SSD), and disk. With the memory tier and the tiered storage architecture, Alluxio can store the important and most frequently accessed data in the fastest memory tier, and store less accessed data in slower tiers with more capacity.

SQL engines can access any data via the Alluxio unified namespace, and the data will be either fetched from the configured mounted storages, or served from Alluxio-managed storage (memory, SSD, and disk), which can greatly improve performance. When Alluxio is deployed colocated with the computation cluster, the Alluxio memory storage can behave similar to the internal caches of SQL engines. Alluxio is able to transparently cache data from various other storage systems, like HDFS. Additionally, since the Alluxio storage is independent from the SQL engines, the cached data can easily be shared between different computation frameworks, thus allowing greater flexibility for deployments.

Apache Ignite HDFS Cache

Apache Ignite (2018) is a distributed platform for in-memory storage and computation. Apache Ignite is a full stack of components which includes a key/value store, a SQL processing engine, and a distributed computation framework, but can be used to cache data for HDFS and SQL engines on Hadoop.

Apache Ignite provides a caching layer for HDFS via the Ignite File System (IGFS). All clients and applications interact with IGFS for data, and IGFS handles reading and writing the



Caching for SQL-on-Hadoop, Fig. 1 Alluxio mounting various storage systems

data to memory on the nodes. When using IGFS as a cache, users can optionally configure a secondary file system for IGFS, which allows transparent read-through and write-through behavior. Users can configure HDFS for the secondary file system, so when the data is synced to IGFS, the data in Ignite is cached in memory.

Using Apache Ignite, SQL engines can access their HDFS data via IGFS, and the data can be cached in the Ignite memory storage. Any SQL engine can take advantage of this cached data for greater performance for queries. When using IGFS, the SQL engine internal caching does not need to be used, which will also enable sharing of the cached data. However, IGFS only allows a single secondary HDFS-compatible file system, if SQL queries need to access varied sources, IGFS would not be able to cache the data.

HDFS Centralized Cache Management

Another external caching option is for the cache to be implemented in the Hadoop storage system, HDFS. Depending on the operating system (OS) buffer cache is one way to take advantage of caching in HDFS. However, there is no user control over the OS buffer cache. Instead, HDFS has a centralized cache management feature (Apache Hadoop HDFS 2018), which enables users to explicitly specify files or directories which should be cached in memory. Once a user specifies a path to cache in HDFS, the HDFS DataNodes will be notified, and will cache the data in off-heap memory. In order to remove a file or directory from the cache, a separate command must be invoked. HDFS caching can help accelerate access to the data, since the data can be stored in memory, instead of disk.

By enabling the HDFS centralized cache, the data of specified files will reside on memory on the HDFS DataNodes, so any SQL engines or applications accessing those files will be able to read the data from memory. Increasing the data access performance of HDFS will help any queries accessing the cached files. Apache Impala has additional support for utilizing the HDFS centralized caching feature for Impala tables which are read from HDFS. With Apache Impala, users can specify tables or partitions of tables to be cached via HDFS centralized caching. IBM Big SQL (Floratou et al. 2016) also uses the HDFS cache for accelerating access to data and utilizes new caching algorithms Adaptive SLRU-K and Adaptive EXD to improve the cache hit rates for better performance.

Since the centralized caching is an HDFS feature, it can help improve performance for any HDFS data access. However, for disaggregated clusters, where the HDFS cluster is separate from the SQL engine cluster, the HDFS caching may not be able to help much. Separating computation and storage clusters is becoming popular for its scalability, flexibility, and cost effectiveness. In these environments, when the computation cluster requires data, the data must be accessed across the network, so even if the data resides in the memory cache on the HDFS DataNodes, the network transfer may be the performance bottleneck. Therefore, further improving performance via caching requires either SQL engine internal caching or a separate caching system. Also, since the caching system is tightly integrated with HDFS, the deployment is not as flexible as other external caching systems.

Conclusion

Caching can significantly improve query performance for SQL engines on Hadoop. There are several different techniques for caching data for SQL frameworks. SQL engine internal caching has the potential for the greatest performance, but each framework must implement a cache, and the cached data is not always shareable by other applications or queries. Another option is to use the HDFS centralized caching feature to store files in memory on the HDFS DataNodes. This server-side cache is useful to speed up local access to data, but if SQL engines are not colocated with HDFS, or the data is not stored in HDFS, the network may become the performance bottleneck. Finally, external systems can cache data for SQL frameworks on Hadoop. When colocated with the computation cluster, external systems can store

data in memory similar to an internal cache, but provide the flexibility to share the data effectively between different applications or queries.

Cross-References

- Apache Spark
- Hadoop
- Hive
- Spark SQL
- Virtual Distributed File System: Alluxio

References

Alluxio (2018) Alluxio – open source memory speed virtual distributed storage. https://www.alluxio.org/. Accessed 19 Mar 2018

Apache Drill (2018) Apache Drill. https://drill.apache.org. Accessed 19 Mar 2018

Apache Hadoop (2018) Welcome to Apache Hadoop! http://hadoop.apache.org. Accessed 19 Mar 2018

Apache Hadoop HDFS (2018) Centralized cache management in HDFS. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html. Accessed 19 Mar 2018

Apache Hive (2018) Apache Hive. https://hive.apache.org/. Accessed 19 Mar 2018

Apache Hive LLAP (2018) LLAP. https://cwiki.apache.org/confluence/display/Hive/LLAP. Accessed 19 Mar 2018

Apache Ignite (2018) Apache Ignite. https://ignite.apache.org/index.html. Accessed 19 Mar 2018

Apache Impala (2018) Apache Impala. https://impala.apache.org/. Accessed 19 Mar 2018

Apache Spark SQL (2018) Spark SQL. https://spark.apache.org/sql/. Accessed 19 Mar 2018

Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113

Facebook (2018) Presto. https://prestodb.io/. Accessed 19 Mar 2018

Floratou A et al (2016) Adaptive caching in big SQL using the HDFS cache. In: SoCC'16 proceedings of the seventh ACM symposium on cloud computing, Snata Clara, 5–7 Oct 2016

Spark RDD (2018) RDD programming guide. http://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence. Accessed 19 Mar 2018

Spark SQL (2018) Spark SQL, dataframes and datasets guide. http://spark.apache.org/docs/latest/sql-programming-guide.html#caching-data-in-memory. Accessed 19 Mar 2018

Cartography

- Visualization

Causal Consistency

- TARDiS: A Branch-and-Merge Approach to Weak Consistency

Certification

- Auditing

Cheap Data Analytics on Cold Storage

Raja Appuswamy
Data Science Department, EURECOM, Biot, France

Definitions

Driven by the desire to extract insights out of data, businesses have started aggregating vast amounts of data from diverse data sources. However, recent analyst reports claim that only 10–20% of data stored is actively accessed with the remaining 80% being *cold* or infrequently accessed. Cold data has also been identified as the fastest-growing storage segment, with a 60% cumulative annual growth rate (Mendoza 2013; Moore 2015; Nandkarni 2014).

As the amount of cold data increases, enterprise customers are increasingly looking for more cost-efficient ways to store this data. A recent report from IDC emphasized the need for such low-cost storage by stating that only 0.5% of potential Big Data is being analyzed, and in order to benefit from unrealized value extraction, infrastructure support is needed to store large volumes of data, over long time duration, at extremely low cost

(Nandkarni 2014). Thus, the topic of managing cold data has received a lot of attention from both industry and academia over the past few years. Systems that are purpose-built to provide cost-efficient storage and analysis of cold data are referred to as *cold storage* systems.

Background and Overview

Enterprise databases have long used storage tiering for reducing capital and operational expenses. Traditionally, databases used a two-tier storage hierarchy. An *online* tier based on enterprise *hard disk drives (HDD)* provided low-latency (ms) access to data. The *backup* tier based on offline tape cartridges or optical drives, in contrast, provided low-cost, high-latency (hours or days) storage for storing backups to be restored only during rare failures.

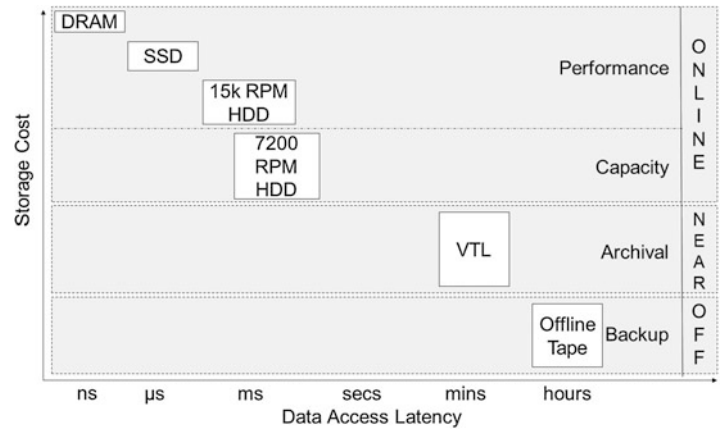
As databases grew in popularity, the necessity to reduce recovery time after failure became important. Further, as regulatory compliance requirements forced enterprises to maintain long-term data archives, the offline nature of the backup tier proved too slow for both storing and retrieving infrequently accessed archival data. This led to the emergence of a new *archival* tier based on nearline storage devices, like virtual tape libraries (VTL), which could store and retrieve data automatically without human intervention in minutes.

Over the past decade, the emergence of flash-based solid-state storage, declining price of DRAM, and demand for low-latency, real-time data analytics have resulted in the traditional online tier being bifurcated into two subtiers, namely, a *performance* tier based on RAM/SSD and a *capacity* tier based on HDD. Thus, most modern enterprises today use a four-tier storage hierarchy (performance, capacity, archival, backup) implemented using three storage types (online, nearline, and offline) as shown in Fig. 1.

Given the proliferation of cold data over the past few years, the obvious question that arises is where should such cold data be stored. Given that databases already have a tiered storage infrastructure in place, an obvious low-cost solution to deal with this problem is to store cold data in either the capacity tier or the archival tier. Unfortunately, both these options are not ideal given recent trends in the high-density storage hardware landscape.

HDD-Based High-Density Storage

Traditionally, 7,200 RPM HDDs have been the primary storage media used for provisioning the capacity tier. For several years, areal density improvements enabled HDDs to increase capacity at Kryder's rate (40% per year), outstripping the 18-month doubling of transistor count predicted by Moore's law. However, over the past few years, HDD vendors have hit walls in scaling areal density with conventional perpendicular magnetic recording (PMR) techniques (Fontana and Decad



Cheap Data Analytics on Cold Storage, Fig. 1 Storage tiering for enterprise databases

2015). As a result, HDD vendors have started working on several techniques to improve areal density, like using helium-bearing instead of air-bearing for the disk head to pack more platters tightly or using shingled magnetic recording (SMR) techniques that pack more data by overlapping adjacent disk tracks, to further increase areal density. Despite such attempts from HDD vendors, HDD areal density is improving only at an annual rate of 16% instead of 40% (Fontana and Decad 2015; Moore 2016).

HDDs also present another problem when used as the storage medium of choice for storing cold data, namely, high idle power consumption. Unlike tape drives, which consume no power once unmounted, HDDs consume a substantial amount of power even while idle (Colarelli and Grunwald 2002). Such power consumption translates to a proportional increase in operational expenses. Thus, the capacity tier, with its always-on HDD-based storage, is not an ideal option for storing infrequently accessed cold data.

Tape-Based High-Density Storage

Unlike HDDs, the areal density of tapes has been increasing steadily at a rate of 33% per year, and the *Linear Tape Organization (LTO)* roadmap (LTO Ultrium 2015) projects continued increase in density for the foreseeable future. Today, a single LTO-7 cartridge is capable of matching or even outperforming a HDD, with respect to sequential data access bandwidth. As modern tape libraries use multiple drives, the cumulative bandwidth achievable using even low-end tape libraries is 1–2 GB/s. However, the random access latency of tape libraries is still 10,000× higher than HDDs (minutes versus ms) due to the fact that tape libraries need to mechanically load tape cartridges before data can be accessed. Thus, tape-based archival tier is used to store only rarely accessed compliance and backup data. As the expected workload in such cases is dominated by sequential writes with a few rare reads, the high access latency of tape drives is tolerable.

Using the archival tier to store cold data, however, changes the application workload, as analytical queries need to be issued over cold data to extract insightful results (EMC 2014). As a nearline storage device with access latency that is five orders of magnitude larger than HDD, tape will impose a significant performance penalty for even latency-insensitive batch analytic workloads (Kathpal and Yasa 2014). Thus, today, enterprises are faced with a trade-off, as they can store cold data in tape-based archival tier at the expense of performance or in the HDD-based capacity tier and trade-off cost.

Key Research Findings

Over the past few years, storage hardware vendors and researchers have become cognizant of the gap between the HDD-based capacity tier and the tape-based archival tier. This has led to the emergence of a new class of nearline storage devices explicitly targeted at cold data workloads. These devices, also referred as *cold storage devices (CSD)*, have two salient properties that distinguish them from the tape-based archival tier. First, they use archival-grade, high-density, shingled magnetic recording-based HDD as the storage media instead of tapes. Second, they explicitly trade off performance for power consumption by organizing hundreds of disks in a massive array of idle disk (MAID) configuration that keeps only a fraction of HDD powered up at any given time (Colarelli and Grunwald 2002).

CSD differ dramatically with respect to price, performance, and peak power consumption characteristics. Some CSD enforce a strict upper limit on the number of HDD that can be spun up at any given point to limit peak power draw. By doing so, they right-provision hardware resources, like in-rack cooling and power management, to cater to the subset of disks that are spun up, reducing operational expenses further. For instance, Pelican (Balakrishnan et al. 2014) packs 1,152 SMR disks in a 52U rack for a total capacity of 5 PB. However, only 8% of disks are spun up at any given time due to restrictions enforced by in-rack cooling and power budget. Similarly, each OpenVault Knox (Yan 2013) CSD stores 30 SMR HDDs in a 2U chassis of which only one can be spun up to minimize the sensitivity

of disks to vibration. Other CSD, like Spectra ArcticBlue Spectra Logic (2013), provide more performance flexibility at the expense of cost, similar to traditional MAID arrays, by only spinning down idle disks while permitting a much larger subset of disks to be active simultaneously.

The net effect of these limitations is that the latency to access data stored in the CSD depends on whether the data resides in a disk that is spun up or down. Access to data in any of the spun-up disks can be done with latency and bandwidth comparable to that of the traditional capacity tier. For instance, Pelican, OpenVault Knox, and ArcticBlue are all capable of saturating a 10 Gb Ethernet link as they provide between 1 and 2 GB/s of throughput with an access latency of 5–10 milliseconds for reading data from spun-up disks. However, accessing data on a spun-down disk takes 5–10 s, as the target disk needs to be spun up before data can be accessed.

CSD form a perfect middle ground between HDD and tape. Due to the use of high-density disks and MAID techniques, CSD are touted to offer cost/GB comparable to tape. With worst-case access latencies in seconds, CSD are closer to HDD than tape with respect to performance. However, CSD are not a drop-in replacement for HDD (Borovica-Gajic et al. 2016). In order to effectively exploit CSD for reducing the cost of cold data storage and analytics, cold storage infrastructures need to design three components of both CSD and analytic engines to work in concert with the shared goal of minimizing the number of disk spin-ups, namely, the CSD storage manager, the CSD I/O scheduler, and the analytic engine's query executor.

CSD Storage Manager

The CSD storage manager is responsible for implementing the storage layout that maps data to disks. Designing an optimal data layout for CSD is a complex problem, as one has to balance performance, reliability, and availability simultaneously. From performance point of view, data that is accessed together should be stored on a set of disks that can spin up together so that requests for such data can be serviced concurrently without any group switches. From the reliability aspect, the disk chosen for storing data should span multiple failure domains, like different trays in the rack, so that a failure in any single domain does not result in total data loss. With respect to availability, data layout should ensure that recovery from a disk failure is seamless and quick, as disk failures will be a norm rather than an exception in CSD.

In CSD like Pelican (Balakrishnan et al. 2014), where the power and cooling infrastructure of the CSD enforces a strict limit on which set of disks can be spun up at any given time, the choice of data layout is a direct consequence of these restrictions. Pelican (Balakrishnan et al. 2014) assembles disks into groups such that disks within each group belong to different failure domains and can be spun up simultaneously. Each data object stored by Pelican is striped across disks within a single group, using erasure coding to protect data in case of disk failures. By using disks that can be active at the same time, Pelican meets the performance requirement, as multiple object requests can be served concurrently due to striping. Erasure coding helps in improving availability, as recovery after disk failures can be done within a group without any spin-ups and several disks can participate in data restoration.

A storage layout that uses historic data access patterns as a hint for future accesses could further improve performance by packing multiple objects that are always accessed together into a few disk groups. The task of identifying an optimal mapping of objects to disks given a history of accesses can be modeled as an optimization problem (Reddy et al. 2015). However, given the large number of objects stored in CSD, it can be quite complicated to handle each object individually in the optimization algorithm. Thus, the problem of choosing a data layout is decomposed into two subproblems: (i) identifying clusters of objects that are accessed together from the trace and (ii) identifying an optimal mapping of objects to disks. The goal of the first step is to transform individual object access patterns into group access patterns by clustering objects that are accessed together. The optimization algorithm can then map clusters to disks. It has been shown that such an access-driven data layout approach can save

up to 78% power over random placement (Reddy et al. 2015).

CSD I/O Scheduler

The goal of the CSD's I/O scheduler is to service object requests from different clients in a way that minimizes the number of group switches while at the same time ensuring fairness across clients. The CSD group scheduling problem can be reduced to the single-head tape scheduling problem in traditional tertiary storage systems, where it has been shown that an algorithm that picks the tape with the largest number of pending requests as the target to be loaded next performs within 2% of the theoretically optimal algorithm (Prabhakar et al. 2003). If efficiency was the only goal, we could apply such an algorithm, which we henceforth refer to as *Max-Requests*, to the CSD case by picking the disk group with the maximum number of requests. However, the algorithm would not provide fairness, as a continuous stream of requests for a few popular disks can starve out requests for less popular ones.

Pelican CSD solves this problem by scheduling object requests in a *first-come first-serve (FCFS)* order to provide fairness with some parameterized slack that occasionally violates the strict FCFS ordering by reordering and grouping requests to the same disk group to improve performance (Balakrishnan et al. 2014). However, by building on FCFS, Pelican's algorithm trades off efficiency for fairness. The Skipper framework (Borovica-Gajic et al. 2016) introduces a new algorithm that strikes a balance between FCFS and Max-Requests by associating a rank with each group that is computed by considering both the number of requests that need to be serviced for that disk group, and the average time spent by requests waiting for that disk group to spin up. The scheduler always picks the group with the highest rank as the target to be spun up next. By using the number of requests in computing the rank, the scheduler behaves efficiently similar to the Max-Requests algorithm. By including the average waiting time, the scheduler ensures that a request to a less popular group that has been not been serviced for a while will get prioritized over other requests to more popular groups.

Database Query Executor

When a CSD is used as the storage back end for an analytical database engine, query execution will result in the database reading data objects directly from the CSD. In the best case, these database read requests are always serviced from a disk group that is spun up. In such a case, there would be no performance difference between using a CSD and the traditional HDD-based capacity tier. However, in the pathological case, every data access request issued by the database would incur a spin-up delay and cripple performance. Unfortunately, the average case is more likely to be similar to the pathological case due to two assumptions made by traditional databases: (1) storage subsystem has exclusive control over data allocation, and (2) the underlying storage media support random accesses with uniform access latency.

Today, most enterprise databases run in a virtualized setting in a private cloud hosted on premise or in the public cloud. In such a virtualized environment, a CSD will store data corresponding to several databases by virtualizing available storage behind an object interface, similar to OpenStack Swift (Oracle 2015) or Amazon S3 (Amazon 2015). As each database reads objects from the CSD, it has no control over the CSD data layout which is performed by the CSD storage manager. The lack of control over data layout implies that the latency to access a set of relations depends on the way they are laid out across disk groups.

Moreover, the CSD services requests from multiple databases simultaneously. Thus, even if all data corresponding to a single database is located in a single group, the execution time of a single query is not guaranteed to be free of disk spin-ups. This is because the access latency of any database request depends on the currently loaded group, which depends on the set of requests from other databases being serviced at any given time. Thus, in a virtualized enterprise data center that uses CSD as a shared service, both assumptions made by analytical database engines are invalidated leading to suboptimal query execution (Borovica-Gajic et al. 2016).

Exploiting the cost benefits of CSD while minimizing the associated performance trade-off requires eliminating unnecessary disk spin-ups. The only way of achieving such an optimal data access pattern is to have the database issue requests for all necessary data upfront so that the CSD can return data back in an order that minimizes the number of disk spin-ups. Thus, the order in which database receives data, and hence the order in which query execution happens, should be determined by the CSD to minimize the performance impact of group switches.

Unfortunately, current databases are not designed to work with such a CSD-driven query execution approach. Traditionally, databases have used a strict optimize-then-execute model to evaluate queries. The database query optimizer uses cost models and statistics gathered to determine the optimal query plan. Once the query plan has been generated, the execution engine then invokes various relational operators strictly based on the generated query plan with no run-time decision-making. This results in pull-based query execution where the database explicitly requests data in an order determined by the query plan. This pull-based execution approach is incompatible with the CSD-driven approach, as the optimal order chosen by the CSD for minimizing group switches is different from the ordering specified by the query optimizer.

In order to enable CSD-driven query execution, analytical database engines should adopt push-based query execution instead of a pull-based one. There are two approaches that can be used to perform push-based query execution. The first approach is to tightly couple query execution with I/O scheduling in such a way that the scheduler knows and delivers only objects that are required for the query executor to make progress. For instance, in a two-table hash join, the scheduler would deliver objects corresponding to the build relation over which the hash table is built before delivering the probe relation. Such an approach was used by tertiary database engines (Sarawagi and Stonebraker 1996) and makes two assumptions: (i) the database has exclusive access to the storage device, and (ii) the I/O scheduler on the storage device is aware of database query execution. Unfortunately, both assumptions are not true today with CSD, as CSD storage is exposed behind an object interface and shared among multiple databases.

The second approach, exemplified by the Skipper framework (Borovica-Gajic et al. 2016), is to perform CSD-driven, push-based, out-of-order execution of queries by building on work done in *adaptive query processing (AQP)* (Deshpande et al, 2007). *Multiway join (MJoin)* (Viglas et al. 2003) is one such AQP technique that enables out-of-order execution by using an n-ary join and symmetric hashing to probe tuples as data arrives, instead of using blocking binary joins whose ordering is predetermined by the query optimizer. However, the incremental nature of symmetric hashing in traditional MJoin requires buffering all input data, as tuples that are yet to arrive could join with any of the already received tuples. Thus, in the worst-case scenario of a query that involves all relations in the dataset, the MJoin cache must be large enough to hold the entire dataset. This requirement makes traditional MJoin inappropriate for the CSD use case as having a buffer cache as large as the entire dataset defeats the purpose of storing data on the CSD. Skipper solves this problem by redesigning both MJoin and database buffer caching algorithms to work in concert such that MJoin can perform query execution even with limited cache capacity (Borovica-Gajic et al. 2016).

Examples of Application

With the right combination of layout, I/O scheduling, and query execution, cold data analytic engines like Skipper have demonstrated that it is possible to mask the long access latency of CSD for latency-insensitive batch analytic workloads. Thus, the obvious application for CSD is as the medium of choice for storing cold data, and a natural way of integrating CSD in the four-tier storage hierarchy shown in Fig. 1 is to use it for building a new cold storage tier that stores only cold data.

However, CSD could potentially have a much larger impact on the storage hierarchy based

on two observations. First, with latency-critical workloads running in the performance tier, the HDD-based capacity tier is already limited to batch analytic workloads today. Second, due to the use of high-density SMR disks and MAID techniques, CSD can offer cost and capacity comparable to the tape-based archival tier. Given these two observations, it might be feasible to get rid of the capacity and archival tiers by using a single consolidated cold storage tier. Such an approach would result in the four-tier hierarchy shown in Fig. 1 being reduced to a three-tier hierarchy with DRAM or SSD in the performance tier, CSD in the cold storage tier, and tape in the backup tier. Recent studies report that enterprises can lower their total cost of ownership (TCO) between 40% and 60% by using such storage consolidation (Borovica-Gajic et al. 2016). In terms of absolute savings, these values translate to hundreds of thousands of dollars for a 100TB database and tens of millions of dollars for larger PB-sized databases.

The implications and benefits of using CSD are also applicable to cloud service providers. For instance, cloud providers have already started deploying custom-built CSD for storing cold data (Bandaru and Patiejunas 2015; Bhat 2016; Google 2017). An interesting area of future work involves exploring implementation and pricing aspects of cloud-hosted "cheap-analytics-over-CSD-as-a-service" solutions. Such services would benefit both customers and providers alike, as providers can increase revenue by expanding their storage-as-a-service offering to include cheap analytic services and customers can reduce the TCO by running latency-insensitive analytic workloads on data stored in CSD.

Future Directions for Research

Systems like Pelican (Balakrishnan et al. 2014) and Skipper (Borovica-Gajic et al. 2016) are a first essential step toward broader adoption of cold data analytics, in general, and CSD in particular. There are several avenues of future work with respect to both cold storage hardware and data analytic software.

Over the past few years, several other systems have been built using alternate storage media to reduce the cost of storing cold data. For instance, DTStore (Lee et al. 2016) uses LTFS tape archive for reducing the TCO of online multimedia streaming services by classifying data based on access pattern and storing cold data in tape drives. ROS (Yan et al. 2017) is a rack-scale optical disk library that provides PB-sized storage with in-line accessibility for cold data using thousands of optical disks packed in a single 42U rack. Nakshatra (Kathpal and Yasa 2014) enables Hadoop-based batch analytic workloads to run directly over data stored in tape archives. Similar to Skipper, Nakshatra also modifies both the tape archive's I/O scheduler and the MapReduce run time to perform push-based execution to hide the long access latency of tape drives.

Today, it is unclear as to how these alternative storage options fare with respect to SMR-HDD-based CSD as the storage media of choice for storing cold data. Some storage media, like optical disks, have the shortcoming that their sequential access bandwidth is substantially lower than HDD or tape. Tape and optical media also have very high access latency compared to CSD. However, an interesting artifact of using push-based engines like Skipper is that query execution becomes much less sensitive to the long access latency of storage devices. Similarly, optical disks are traditionally known to have longer life span compared to HDD and tape. However, it is possible to deal with even frequent media failures by trading off some storage capacity for improved reliability by using erasure coding and data replication techniques in software. Thus, further research is required to understand the implications of storing cold data on these storage devices by running realistic cold data analytic workloads and performing a comparative study of performance and reliability.

On the software side, current systems like Skipper and Nakshatra only target a simplified subset of SQL or MapReduce-based analytic workloads. They do not support complex business intelligence workloads that require operators like data cube (Gray et al. 1997), data mining workloads like clustering and

classification, or iterative machine learning workloads. They also do not consider the availability of additional layers of persistent storage that could be used as caches or prefetch buffers for data stored in CSD or as an extended memory that can be used to swap out intermediate data structures generated by operators like MJoin. Further research is also required to understand how push-based execution can be extended to these workloads.

References

Amazon (2015) Amazon simple storage service. https://aws.amazon.com/s3/. Accessed 1 Oct 2017

Appuswamy R, Borovica-Gajic R, Graefe G, Ailamaki A (2017) The five-minute rule thirty years later, and its impact on the storage hierarchy. In: Proceedings of the eighth international workshop on accelerating analytics and data management systems using modern processor and storage architectures, Munich

Balakrishnan S, Black R, Donnelly A, England P, Glass A, Harper D, Legtchenko S, Ogus A, Peterson E, Rowstron A (2014) Pelican: a building block for exascale cold data storage. In: Proceedings of the 11th USENIX conference on operating systems design and implementation, Berkeley, pp 351–365

Bandaru K, Patiejunas K (2015) Under the hood: Facebooks cold storage system. Facebook. https://code.facebook.com/posts/1433093613662262/-under-the-hood-facebook-s-cold-storage-system-/. Accessed 1 Oct 2017

Bhat S (2016) Introducing azure cool blob storage. Microsoft. https://azure.microsoft.com/en-us/blog/introducing-azure-cool-storage/. Accessed 1 Oct 2017

Borovica-Gajic R, Appuswamy R, Ailamaki A (2016) Cheap data analytics using cold storage devices. Proc VLDB Endow 9(12):1029–1040. https://doi.org/10.14778/2994509.2994521

Colarelli D, Grunwald D (2002) Massive arrays of idle disks for storage archives. In: Proceedings of the ACM/IEEE conference on supercomputing, Los Alamitos, pp 1–11

Deshpande A, Ives Z, Raman V (2007) Adaptive query processing. J Found Trends Databases 1(1):1–140. https://doi.org/10.1561/1900000001

EMC (2014) The digital universe of opportunities: rich data and the increasing value of the internet of things. https://www.emc.com/collateral/analyst-reports/idc-digital-universe-2014.pdf. Accessed 1 Oct 2017

Fontana R, Decad G (2015) Roadmaps and technology reality. Presented at the library of congress storage meetings on designing storage architectures for digital collections, Washington, 9–10 Sept 2015

Google (2017) Nearline cloud storage. https://cloud.google.com/storage/archival/. Cited 1 Oct 2017

Gray J, Graefe G (1997) The five-minute rule ten years later, and other computer storage rules of thumb. SIGMOD Rec 26(4):63–68. https://doi.org/10.1145/271074.271094

Gray J, Graefe G (2007) The five-minute rule twenty years later and how flash memory changes the rules. In: Proceedings of the 3rd international workshop on data management on new hardware, New York, pp 1–9

Gray J, Putzolu GR (1987) The 5-minute rule for trading memory for disk accesses and the 10-byte rule for trading memory for CPU time. SIGMOD Rec 16(3):395–398. https://doi.org/10.1145/38714.38755

Gray J, Chaudhuri S, Bosworth A, Layman A, Reichart D, Venkatrao M, Pellow F, Pirahesh H (1997) Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. J Data Min Knowl Discov 1(1):29–53. https://doi.org/10.1023/A:1009726021843

Kathpal A, Yasa GAN (2014) Nakshatra: towards running batch analytics on an archive. In: Proceedings of 22nd IEEE international symposium on the modelling, analysis and simulation of computer and telecommunication systems, Paris, pp 479–482

Lee J, Ahn J, Park C, Kim J (2016) DTStorage: dynamic tape-based storage for cost-effective and highly-available streaming service. In: Proceedings of the 16th IEEE/ACM international symposium on cluster, cloud and grid computing, Cartagena, pp 376–387

LTO Ultrium (2015) LTO roadmap. http://www.ltoultrium.com/lto-ultrium-roadmap/. Accessed 1 Oct 2017

Mendoza A (2013) Cold storage in the cloud: trends, challenges, and solutions. https://www.intel.com/content/www/us/en/storage/cold-storage-atom-xeon-paper.html. Accessed 1 Oct 2017

Moore F (2015) Tiered storage takes center stage. Horison Inc. http://horison.com/publications/tiered-storage-takes-center-stage. Accessed 1 Oct 2017

Moore F (2016) Storage outlook 2016. Horison Inc. https://horison.com/publications/storage-outlook-2016. Accessed 1 Oct 2017

Nandkarni A (2014) IDC worldwide cold storage taxonomy. http://www.idc.com/getdoc.jsp?containerId=246732. Accessed 1 Oct 2017

Oracle (2015) OpenStack swift interface for oracle hierarchical storage manager. http://www.oracle.com/us/products/servers-storage/storage/storage-software/solution-brief-sam-swift-2321869.pdf. Accessed 1 Oct 2017

Prabhakar S, Agrawal D, Abbadi A (2003) Optimal scheduling algorithms for tertiary storage. J Distrib Parallel Databases 14(3):255–282. https://doi.org/10.1023/A:1025589332623

Reddy R, Kathpal A, Basak J, Katz R (2015) Data layout for power efficient archival storage systems. In: Proceedings of the workshop on power-aware computing and systems, Monterey, pp 16–20

Robert Y, Vivien F (2009) Introduction to scheduling, 1st edn. CRC Press, Boca Raton

Sarawagi S, Stonebraker M (1996) Reordering query execution in tertiary memory databases. In: Proceedings of the 22th international conference on very large data bases, San Francisco, pp 156–167

Spectra Logic (2013) Spectra arcticblue overview. https://www.spectralogic.com/products/arcticblue/. Accessed 1 Oct 2017

Viglas SD, Naughton JF, Burger F (2003) Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings of the 29th international conference on very large data bases, Berlin, pp 285–296

Yan M (2013) Cold storage hardware v0.5. http://www.opencompute.org/wp/wp-content/uploads/2013/01/Open_Compute_Project_Cold_Storage_Specification_v0.5.pdf. Accessed 1 Oct 2017

Yan W, Yao J, Cao Q, Xie C, Jiang H (2017) ROS: a rack-based optical storage system with inline accessibility for long-term data preservation. In: Proceedings of the 12th European conference on computer systems, Belgrade, pp 161–174

Clojure

Nicolas Modrzyk
Karabiner Software, Tokyo, Japan

> It's easy to get sidetracked with technology, and that is the danger, but ultimately you have to see what works with the music and what doesn't. In a lot of cases, less is more. In most cases, less is more.
>
> Herbie Hancock

Definitions

How to use some best features of Clojure, a LISP like functional programming language, to target big data analysis.

Introduction

Rich Hickeys Clojure language is not in the new section of hyped languages anymore, which is great news for everybody interested in Clojure. The language has just reached version 1.9, and the amount of strong and precise development that went into this new version deserved an ovational praise. But why would you use Clojure nowadays? What can it do for you that the available trillions of other languages available on the market cannot do, or not as efficiently, with programming in Java 1.9, Python, or NodeJS getting all the hype and the tooling nowadays, with even big consulting firms entering the NodeJS market. Clojure being a LISP has a fantastic minimal core and a very small set of functions and data structures. Of course, you have asynchronous thread-safe parallel queues, of course you have parallel processing of all the data structures, and it is also a breeze to go for test-driven development, but what's more is that Clojure changes the way you think about programming and write programs in general. It makes you think in small, versatile, reusable building block. Eventually the users of Clojure, the people writing code with it, do not really think about it anymore; it just becomes the way they naturally code, whatever the language. Clojure also makes your code smaller, easier to debug, maintain, and push to production with less fear of being woken up at 3 AM for issues. Your program is consistent and reliable. Let's review some of the Clojure features and ecosystem that makes Clojure a language of choice for handling big data.

Clojure for Instant Prototyping, with a REPL

Many languages have a REPL nowadays, a Read-Eval-Print-Loop, like a shell prompt, where you can write and execute code block by block. Clojure is not the only language armed with a REPL, but it is one of the most convenient because many functions consist of a simple code block that usually holds on one line.

Take the Fibonacci sequence, for example, it does hold as a one liner

```
(def fib-seq
  ((fn rfib [a b]
     (lazy-seq
       (cons a
         (rfib b (+ a b)))))
     0 1))
```

and is still easily readable.

You usually start a REPL with Leiningen (or boot), the two most famous build tools that actually install Clojure at the same time and make you in command of a build environment at the same time.

Starting a REPL with:

```
lein repl
```

or

```
boot repl
```

And you can copy paste the code above and retrieve the first n elements of the Fibonacci sequence, with the result printed below.

```
(take 10 fib-seq)
;(0 1 1 2 3 5 8 13 21 34)
```

When working with Clojure, programmers would usually always have a REPL running to try ideas on the run or in a business meeting.

Clojure for Code as Data

Beyond having a REPL, Clojure being a LISP has its syntax and its code execution commands virtually identical. Clojure code is made of successive lists of lists of symbols.

What does that mean?

At the REPL, since you have it open, if you want to perform the simple Math of adding 1 to 1, you would write: (remember the LISP prefix notation ...)

```
(+ 1 1)
; that returns 2
```

The code is made of a single sequence, surrounded by parenthesis, and the sequence itself is made of 3 characters, +, 1, and 1.

What the REPL does is converting the string you have written into a sequence and then evaluating its content. To perform the same thing the REPL is doing, you can use the function read-string, where read-string takes a string and converts it to a sequence of symbols, similar to an abstract syntax tree.

```
(read-string "(+ 1 1)")
; (+ 1 1)
```

To evaluate the list, you call eval on the list, which executes the code and returns 2 again.

```
(eval
  (read-string "(+ 1 1)"))
; 2
```

To run eval on a sequence directly, people usually use the ‘ (quote) symbol, where the quote means this is already a set of symbols as a sequence, no need to read it.

```
(eval '(+ 1 1))
; 2
```

Why is this nice? It makes it really easy to tweak the compilation step found in coding. To achieve that, Clojure provides macros, which are evaluated during the pre-processing phase, transforming the lists of lists of sequence before they are executed. See the plus-one macro below, which adds one to the variable x.

```
(defmacro plus-one [x]
   '(+ 1 ~x))
```

When performing the compilating step of eval, this macro will rewrite the list as a list of 3 symbols, as seen above, and as expanded below:

```
(clojure.pprint/pprint
  (macroexpand '(plus-one 1)))
; (clojure.core/+ 1 1)
```

This is different than writing a simple function, plus-one-fn:

```
(defn plus-one-fn [x] (+ 1 x))
```

In the sense, that the block code is written, and evaluated as is, not rewritten then evaluated. Which takes us to ...

Clojure Threading Macros

Clojure Macros have the ability to rewrite the sequence of lists of lists before they are executed. Two of the most used macros are `->` and `->>`. The first macro, `->`, takes a list of forms and rewrite in a different order. The new order is such that the previous step block comes as the first parameter, and so the second element of the list in the next step. With the simple (+ 1 1) example, again this gives:

```
(-> 1
  (+ 1))
; 2
```

The rewritten code can be expanded and seen using Clojure's macroexpand:

```
(macroexpand '(-> 1 (+ 1)))
; (clojure.core/+ 1 1)
```

Those two macros come very handy when handling data. Let's take a new example with `->>`, the macro that rewrites code with the previous code block becoming the last element of next steps list. Say you want to count the sum of all the integers from 1 to 100000, with the `->>` macro, this would be written:

```
(->> 100000
  (range)
  (apply +))
```

or as a one liner:

```
(->> 100000 (range) (apply +))
```

Again, using macroexpand, you would notice this expands to:

```
(apply + (range 100000))
```

But was much more readable. An example of a more complex operation, where:

- a sequence from 0 up to 100000 is created,
- then each element of the sequence is incremented,
- then each element of the sequence is incremented,
- and finally create a vector of the result

The above processing could be written like this:

```
(into []
  (filter
   even?
   (map inc (range 100000))))
```

Or the definitely more readable:

```
(->>
  100000
  (range)
  (map inc)
  (filter even?)
  (into []))
```

Those Clojure threading macros tend to effectively be usable in many places, For example, in the Origami library, which is a Clojure wrapper around OpenCV, transformations can be applied to images in a sequence also using the `->` macro, as shown below:

```
(->
 (imread "doc/cat_in_bowl.jpeg")
 (cvt-color! COLOR_RGB2GRAY)
 (canny! 300.0 100.0 3 true)
 (bitwise-not!)
 (u/resize-by 0.5)
 (imwrite "doc/canny-cat.jpg"))
```

The result of the transformation shown in the pictures below (Figs. 1 and 2).

Clojure Laziness and Reducers

Usually Clojure tries to do the least amount of work as possible. Meaning, lists of lists are lazily evaluated, effectively evaluated only when really required. Laziness is a core feature of Clojure and

Clojure, Fig. 1 Cat in a bowl

Clojure, Fig. 2 Cat in a bowl

of many programmers. In the example below, a binding named lazy-var is created.

```
(def lazy-var (plus-one-fn 3))
```

The binding was created, but it was not really used yet. It will only be used when, for example, it is required to be printed.

```
(println lazy-var);
 4
```

Laziness is used in many functions of Clojure core, so those functions keep a good balance between efficiency and laziness. (Isn't that what we are all trying to achieve anyway?) Reducers, or reducing functions, on the contrary know they will be combined to be applied on collections that will finally turn into a realized result, realized taken in the sense of not-lazy.

And so, knowing this, reducers can efficiently work in parallel across all the elements of collections they are being applied to. Here is the simple threading example again, this time using reducers.

```
(require
  '[clojure.core.reducers :as r])

(->>
 100000
 (range)
 (r/map inc)
 (r/filter even?)
 (into []))
```

It looks pretty much the same on paper, apart from the r/ prefixes added to the map and filter functions. What does not show on paper is the fact that the r/map and r/filter functions have been running in parallel and, on sizeable collections, the time required to evaluate all the forms is substantially faster.

The claypool library, while not an official Clojure library, is quite convenient to specify the size of the different thread pools used to do parallel processing on collections.

In Clojure, there is an infamous function named pmap that has been known to make great use of all the cores available to the runtime. (And take down clusters by using all the possible resources available.)

In Clojure, to create a sequence of five elements from 0 to 4 and increase all the elements sequentially, map is usually used.

```
(->>
 5
 (range)
 (map inc))
```

To perform the same processing using as many cores as possible, its parallel version named pmap is usually used.

```
(->>
  5
  (range)
  (pmap inc))
```

And to perform the same parallel work again, but using only four cores, the claypoole library version of pmap could be used, this time specifying the thread pool size.

```
(require
  '[com.climate.claypoole :as cp])

(->>
  5
  (range)
  (cp/pmap 4 inc))
```

Clojure for Multi-threading

Reducers and the different versions of pmap are perfect to apply computations in parallel. To perform asynchronous message handling between different blocks of code, Clojure comes with a core library named core.async, available on github.

To present this, a small example made of two asynchronous blocks of code, exchanging messages through a channel will be used.

One asynchronous block will send messages to the channel, while the other will read messages from that same channel and print them out.

This very first example creates a thread and prints message asynchronously by itself.

```
(require
  '[clojure.core.async
      :refer :all])

(thread
 (dotimes [n 10]
   (Thread/sleep 100)
   (println "hello:" n)))
```

If executed at the REPL, as soon as the block of code is evaluated, evaluation does not block and returns a reference to the executing async thread.

In the meantime, the thread goes on with his happy life and starts printing hello messages every 100 ms.

This second example creates two threads. A thread will update an atom, a variable ready for asynchronous updates, by safely increasing its value by one. A second thread safely reads the value of the atom, while it's being updated by the first thread.

```
(def my-value (atom 0))

(thread
 (dotimes [n 10]
   (Thread/sleep 1000)
   (swap! my-value inc)))

 (thread
 (dotimes [_ 5]
   (Thread/sleep 1000)
   (println
     "Counter is now:"
     @my-value)))
```

The number of threads can be increased, and in action, no deadlock occurs. Updates to the atom are done using the STM transaction model that prevents deadlock to occur using transaction to perform updates on atoms.

The third asynchronous example removes the use of the intermediate atom, by directly acting on messages sent via a channel.

The first thread still creates messages, but this time sends them to the newly created hi-chan channel, while the second thread reads the value coming through the channel and prints them out.

```
(def hi-chan (chan))

(thread
 (dotimes [n 10]
   (Thread/sleep 1000)
   (>!! hi-chan
        (str "hello:" n))))

(thread
 (dotimes [_ 10]
   (Thread/sleep 1000)
   (println
      "Channel value:"
      (<!! hi-chan ))))
```

Here again, both thread code blocks return immediately without blocking the execution of the main thread, making all this feasible to execute at the REPL.

Core.async can also run in the browser via ClojureScript and is a very compelling strategy to create a back buffer for onscreen user events and other asynchronous data requests.

Clojure for the Backend

Many frameworks exist for server and api development in Clojure, but most of them seem to be inspired or paying close attention to the efforts that were put into the ring and compojure mini-frameworks.

Ring itself is an abstraction over HTTP, while Compojure is a routing library that can, and mostly does, run on top of ring.

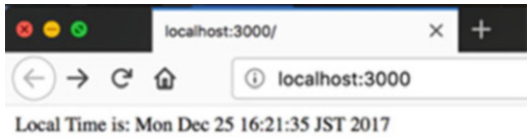
If you have Leiningen installed, getting started with Compojure is as easy as using the compojure project template.

```
lein new compojure intro
```

This creates a prod-ready but also development-ready environment to start writing code.

From the newly created intro folder, a ring development server can be started using the following command.

```
lein ring server
```



Clojure, Fig. 3 Ring HTTP get

This will set up everything needed, apart from your favorite text editor. The main core.clj file in the created project looks deceptively simple, with a single route defined for a GET request on the root context, anything else returning a simple Not Found string.

```
(defroutes app-routes
  (GET "/" [] "Hello World")
  (route/not-found "Not Found"))

(def app
  (wrap-defaults
    app-routes
    site-defaults))
```

The main route can be updated by adding time via a Java date object, using basic Clojure/Java Interop.

```
(defroutes app-routes
  (GET "/" []
       (str
         "Server Time is: "
         (java.util.Date.)))
  (route/not-found "Not Found"))
```

This is of course nowhere near a production-ready app, but the point here is that this creates a fully ready development environment in a few minutes, thus bringing server-side prototyping setup down to pretty much zero (Fig. 3).

To get further into the backend programming in Clojure, there is a nice introduction in the blog post below, when you want to quickly grasp all the ring/compojure concepts:

sinatra-docs-in-clojure

Clojure for the Front End

ClojureScript is Clojure ported to javascript runtime, and why and how to use it is extensively covered in the Reactive with ClojureScript apress book.

The sheer happiness that results from using ClojureScript for the frontend is that this makes your full application code to be written all in Clojure:

- the backend,
- the frontend, and
- the data exchanged between both the backend and the frontend can also be in the Extensible Data Notation, a subset of Clojure.

Clojure's Reagent is a library that acts as a thin wrapper around React, the frontend framework created by Facebook. It removes the extra javascript boiler-plate to make the code very compact, easier to read, write, and maintain.

Reagent can be combined with, among many others, core.async so to get asynchronous looking code (remember only JavaScript thread in the browser) without sacrificing readability.

As for Clojure on the backend, here again there is a Leiningen project template for the reagent framework that can be created with the command below.

```
lein new reagent reagent-intro
```

To get started with the reagent setup, this Leiningen needs two commands to be started, one for the backend and one for the frontend.

Here is the command to start the server development environment:

```
lein ring server
```

Here is the command to start the client development environment:

```
lein figwheel
```

The setup in place, it is now possible to start editing the core.cljs file in the created project.

```
(defn some-component []
  [:div
   [:h3 "I am a component!"]
   [:p.someclass
    "I have " [:strong "bold"]
    [:span {:style {:color "#cc77cc"}}
    " and red"] " text."]])

(defn mount-root[]
  (reagent/render
    [some-component]
```

```
    (.getElementById js/document
    "app")))

(defn init![]
  (mount-root))
```

The code above renders to HTML and is returned to the client as shown in the figure below (Fig. 4).

init! is defined and called only once in development mode; thus the mount-root, which can be rewritten, is kept separate.

The function mount-root itself takes a reagent component and a mount point in the current HTML dom, here < div id = app >< /div >, and puts the component at that place.

A component is defined as a standard Clojure function, whose return value is a special kind of markup specific to Clojure, named hiccup.

Hiccup is eventually taken and transformed from a list of Clojure vectors to valid HTML.

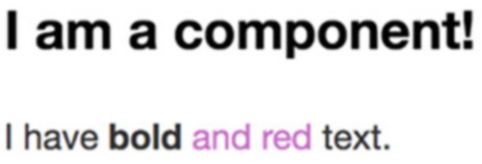
Here the component some-component output some basic HTML elements like div, h3, p, and span elements.

Components can be combined together, so it is possible to create a component from two smaller components, thus being completely in line with Clojure big philosophy of building blocks.

The code below creates a simple Reagent component top-component, made of two components of type some-component.

```
(defn top-component []
  [:div
   [some-component]
[some-component]])
```

Note here that even though some-component is created using a single common definition, the two mounted some-component are different instances.



Clojure, Fig. 4 Reagent

Also, if not obvious, it is not required to have Clojure on the backend to be able to use ClojureScript and Reagent.

Many now in production projects did a rewrite of their frontend from JavaScript to ClojureScript first, to improve maintainability, without updating the server-side code.

Clojure for Big Data

Sparkling and Flambo are two among the Clojure offers to handle spark data and spark clusters.

Flambo is slightly easier and more importantly makes use of the previously introduced threading macros, − > and − >>.

A simple bang-in example of using Flambo is shown below. The example counts the total number of characters of each line, line by line, and then reduces to a single value of the overall total over all the lines.

```
(->
 (f/text-file sc "data.txt")
 ;; returns an unrealized
 ;; lazy dataset
 (f/map (f/fn [s] (count s)))
 ;; returns RDD array
 ;; of length of lines
 (f/reduce (f/fn [x y] (+ x y))))
 ;; use an inline anonymous
 ;; function for reduction
```

Here, sc is a Spark Context, which can target a local in-memory cluster, a stand-alone spark cluster, or a mesos cluster.

```
(def c
 (-> (conf/spark-conf)
     (conf/master "local")
     (conf/app-name "infinity")))

(def sc (f/spark-context c))
```

To perform a spark job, a spark resilient distributed dataset is used, and in the first example presented, the dataset was created by reading from a text file from the local file system.

Creating a Spark RDD can also be done in a few other different ways, like a RDD made from blobs of files stored in HDFS.

```
(def data
(f/text-file
  sc
  (str
```

```
  "hdfs://hostname:port"
  "/home/data"
  "/*/*.bz2"))
```

Of course, using regular Java Interop, it is also possible to target other RDD sources like Cassandra, HBase, and any other storage supported by hadoop.

Flambo itself allows to create valid Spark job that can be deployed and run as any other jobs, but one main limitation it has is that it does not do allow to write and run new job at the REPL without using AOT.

AOT is ahead-of-time compilation meaning you need to statically compile all your code before running, which does prevent it from being created and run at the REPL.

This limitation sparked, pun intended, a new project worth considering named powderkeg, whose goal is to overcome that very limitation.

Clojure for Machine Learning

To go beyond simple big data, and as a recent and strong contender to the TensorFlow framework, Cortex has recently joined the Clojure ecosystem for machine learning.

Some starters and favorites examples to look at to fully understand how Cortex works are: The cats and dogs sorting example or the fruits classification example.

In this short entry, we will present briefly training a network for xor operations, as it has recently been included in the cortex examples section and is very simple to go over.

As known, xor operations take two values in input and one value in output. This is represented here by creating a dataset to train the network, made of a vector of elements, where each element is a map with an input, :x, and an output, :y, value.

```
(def xor-dataset
  [{:x [0.0 0.0] :y [0.0]}
   {:x [0.0 1.0] :y [1.0]}
   {:x [1.0 0.0] :y [1.0]}
   {:x [1.0 1.0] :y [0.0]}])
```

The neural network to be created will be a linear network of three layers, where the activation layer will be the hyperbolic tangent activation function. (See: tanh function)

In the definition of the network, input and output layer each specify how many variables are available, and so below:

- 2 for the input, and
- 1 for the output

The network is then created almost by translating written text to a network definition using the cortex api.

```
(def nn
 (network/linear-network
  [(layers/input 2 1 1 :id :x)
   (layers/linear->tanh 10)
   (layers/linear 1 :id :y)]))
```

Now along with the dataset, let's create a trained version of this neural network.

In machine learning, training a network is usually done using two data sets.

- a training set, to tell the network what is expected and correct
- a testing set, to test the network and check whether how it performs.

The training stage is then achieved by passing those two data sets, but here the same dataset is used.

The number of rounds of training is also specified, here 3000 to get an efficient network.

```
(def trained
  (train/train-n
       nn
       xor-dataset
       xor-dataset
    :batch-size 4
    :epoch-count 3000
    :simple-loss-print? false))
```

Each training step outputs some output of the current state of the trained network, notably whether it performed better or not than its own version in the previous step.

```
Training network:
...
Loss for epoch 1:
  (current) 0.7927149 (best)
Saving network to
  trained-network.nippy
```

Finally, to validate results yet another dataset is created, of usually new values, and of course only input values.

```
(def new-dataset
  [{:x [0.0 0.0] }
   {:x [0.0 1.0] }
   {:x [1.0 0.0] }
   {:x [1.0 1.0] }])
```

Then follows running the trained network with this new dataset.

```
(clojure.pprint/pprint
  (execute/run
    trained
    new-dataset))
```

The result is, not surprisingly, very accurate, as shown from a screenshot of the network executed at the REPL.

```
[{:y [0.4397728145122528]}
 {:y [0.3689133822917938]}
 {:y [0.37379032373428345]}
 {:y [0.4282912313938141]}]
```

Clojure for Teaching

Another small in size but very important contribution to the Clojure ecosystem is named Gorilla.

Simply said, Gorilla is a web-based REPL, which allows to create annotated NoteBooks, along ready to be executed code blocks, all in the browser.

It is made of a server-side part that runs a background repl and a simple compojure-based API ready to receive Clojure code as string.

The second part is a Clojure frontend, running a websocket sending Clojure code to the REPL on the server, and waits for the result of the execution and displays it.

The gorilla framework can be added to any of your project using a Leiningen plugin.

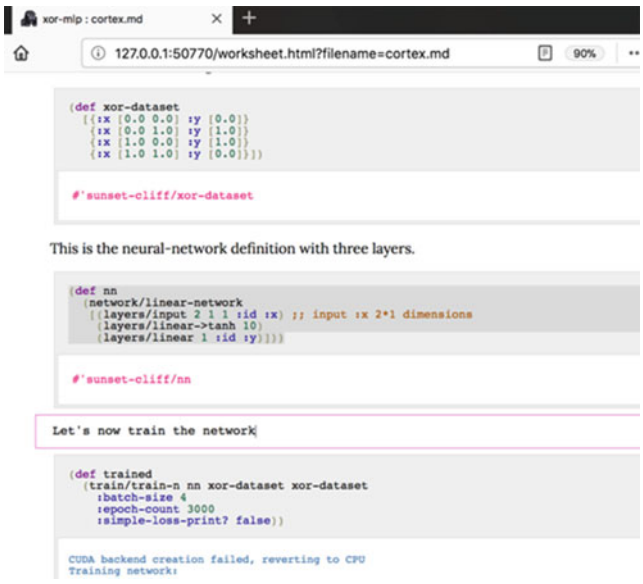
This plugin is added either to the project definition project.clj or the global leiningen profile, profile.clj.

```
[lein-gorilla "0.4.0"]
```

Starting the gorilla environment can now be done using a Leiningen command:

```
lein gorilla
```

Gorilla now makes the library and source code available through a web interface. All the presented Cortex code could be run in the browser, for example, and a possible figure is shown below (Fig. 5).



Clojure, Fig. 5 Gorilla

Beyond

Other key Clojure-based projects that can also be looked into are:

- Riemann is an event aggregator, monitoring system entirely written in Clojure. It can receive and process events from almost anything you can imagine.
- Apache Storm almost entirely written in Clojure, reportedly clocks millions of tuples processed per second per node. This is a framework of choice, and if you have complete decision over the technologies used in your next project, comes as a no brainer.
- Datomic, which is a distributed persistence and query model over an historical set of stored facts.

Conclusion

This short entry presented diverse possibilities on how to put Clojure straight into action to:

- Help developers in writing code using a REPL, thus reducing usual life cycle overhead associated with many programming languages
- Reduce maintenance associated with writing code in other languages, by considerably reducing the lines of code used to work and transform data structure by writing macros and/or using the already available threading macros
- Enable pinpoint custom usage of the cores available to each runtime node of your application. Parallel computations is available to most data structures
- core.async allows message passing between different components of your application. Point of interaction can be defined with atom, or channels, and by being reliably highlighted, reduces the coupling between different parts of the code, making the overall end product resistant to changes.
- Front-to-End Clojure brings commonness and portability of the application code thus helping avoid usual pitfalls when doing client-server programming, in simple sync-ed HTTP requests or real-time websockets.
- Also, not only the code of the application but the projects metadata used by the build tool Leiningen itself was a DSL written in Clojure.
- Bringing entry points to directly interact with any Spark cluster via the REPL, thus making it easier to debug common and more advanced distributed data problems.
- It is also ready for machine learning, providing a real and powerful alternative to TensorFlow with Cortex.
- Finally, the ability to consume all the above, and present annotated pages of work, ready to be distributed and ready for reuse via Gorilla Notebook-style environment.

Cloud Big Data Benchmarks

▸ Virtualized Big Data Benchmarks

Cloud Computing for Big Data Analysis

Fabrizio Marozzo and Loris Belcastro
DIMES, University of Calabria, Rende, Italy

Definitions

Cloud computing is a model that enables convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction (Mell and Grance 2011).

Overview

In the last decade, the ability to produce and gather data has increased exponentially. For ex-

ample, huge amounts of digital data are generated by and collected from several sources, such as sensors, web applications, and services. Moreover, thanks to the growth of social networks (e.g., Facebook, Twitter, Pinterest, Instagram, Foursquare, etc.) and the widespread diffusion of mobile phones, every day millions of people share information about their interests and activities. The amount of data generated, the speed at which it is produced, and its heterogeneity in terms of format represent a challenge to the current storage, process, and analysis capabilities. Those data volumes, commonly referred as Big Data, can be exploited to extract useful information and to produce helpful knowledge for science, industry, public services, and in general humankind.

To extract value from such data, novel technologies and architectures have been developed by data scientists for capturing and analyzing complex and/or high velocity data. In general, the process of knowledge discovery from Big Data is not so easy, mainly due to data characteristics, such as size, complexity, and variety, that are required to address several issues. To overcome these problems and get valuable information and knowledge in shorter time, high-performance and scalable computing systems are used in combination with data and knowledge discovery techniques.

In this context, Cloud computing has emerged as an effective platform to face the challenge of extracting knowledge from Big Data repositories in limited time, as well as to provide an effective and efficient data analysis environment for researchers and companies. From a client perspective, the Cloud is an abstraction for remote, infinitely scalable provisioning of computation and storage resources (Talia et al. 2015). From an implementation point of view, Cloud systems are based on large sets of computing resources, located somewhere "in the Cloud," which are allocated to applications on demand (Barga et al. 2011). Thus, Cloud computing can be defined as a distributed computing paradigm in which all the resources, dynamically scalable and often virtualized, are provided as services over the Internet. As defined by NIST (National Institute of Standards and Technology) (Mell and Grance 2011), Cloud computing can be described as: "*A model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*" From the NIST definition, we can identify five essential characteristics of Cloud computing systems, which are (i) on-demand self-service, (ii) broad network access, (iii) resource pooling, (iv) rapid elasticity, and (v) measured service.

Cloud computing vendors provide their services according to three main distribution models:

- *Software as a Service (SaaS)*, in which software and data are provided through Internet to customers as ready-to-use services. Specifically, software and associated data are hosted by providers, and customers access them without the need to use any additional hardware or software.
- *Platform as a Service (PaaS)*, in an environment including databases, application servers, development environment for building, testing, and running custom applications. Developers can just focus on deploying of applications since Cloud providers are in charge of maintenance and optimization of the environment and underlying infrastructure.
- *Infrastructure as a Service (IaaS)*, that is an outsourcing model under which customers rent resources like CPUs, disks, or more complex resources like virtualized servers or operating systems to support their operations. Compared to the PaaS approach, the IaaS model has a higher system administration costs for the user; on the other hand, IaaS allows a full customization of the execution environment.

Key Research Findings

Most available data analysis solutions today are based on open-source frameworks, such as

Hadoop (https://hadoop.apache.org/) and Spark (https://spark.apache.org/), but there are also some proprietary solutions proposed by big companies (e.g., IBM, Kognitio).

Concerning the distribution models of Cloud services, the most common ones for providing Big Data analysis solutions are PaaS and SaaS. Usually, IaaS is not used for high-level data analysis applications but mainly to handle the storage and computing needs of data analysis processes. In fact, IaaS is the more expensive distribution model, because it requires a greater investment of IT resources. On the contrary, PaaS is widely used for Big Data analysis, because it provides data analysts with tools, programming suites, environments, and libraries ready to be built, deployed, and run on the Cloud platform. With the PaaS model, users do not need to care about configuring and scaling the infrastructure (e.g., a distributed and scalable Hadoop system), because the Cloud vendor will do that for them. Finally, the SaaS model is used to offer complete Big Data analysis applications to end users, so that they can execute analysis on large and/or complex datasets by exploiting Cloud scalability in storing and processing data.

As outlined in Li et al. (2010), users can access Cloud computing services using different client devices, Web browsers, and desktop/mobile applications. The business software and user data are executed and stored on servers hosted in Cloud data centers, which provide storage and computing resources. Such resources include thousands of servers and storage devices connected to each other through an intra-Cloud network.

Several technologies and standards are used by the different components of the architecture. For example, users can interact with Cloud services through SOAP-based or RESTful Web services (Richardson and Ruby 2008) and Ajax technologies, which let Cloud services to have look and interactivity equivalent to those provided by desktop applications.

Developing Cloud-based Big Data analysis applications may be a complex task, with specific issues that go beyond those of stand-alone application programming. For instance, Cloud programming must deal with deployment, scalability, and monitoring aspects that are not easy to handle without the use of ad hoc environments (Talia et al. 2015). In fact, to simplify the development of Cloud applications, specific development environments are often used. Some of the most representative Cloud computing development environments currently in use can be classified into four types:

- *Integrated development environments*, which are used to code, debug, deploy, and monitor Cloud applications that are executed on a Cloud infrastructure, such as Eclipse, Visual Studio, and IntelliJ.
- *Parallel-processing development environments*, which are used to define parallel applications for processing large amount of data that are run on a cluster of virtual machines provided by a Cloud infrastructure (e.g., Hadoop and Spark).
- *Workflow development environments*, which are used to define workflow-based applications that are executed on a Cloud infrastructure, such as Swift and DMCF.
- *Data-analytics development environments*, which are used to define data analysis applications through machine learning and data mining tools provided by a Cloud infrastructure. Some examples are Azure ML and BigML.

The programming model is a key factor to be considered for exploiting the powerful features of Cloud computing. MapReduce (Dean and Ghemawat 2004) is widely recognized as one of the most important programming models for Cloud computing environments, being it supported by Google and other leading Cloud providers such as Amazon, with its Elastic MapReduce service, and Microsoft, with its HDInsight, or on top of private Cloud infrastructures such as OpenNebula, with its Sahara service. Hadoop is the best-known MapReduce implementation, and it is commonly used to develop parallel applications that analyze big amounts of data on Clouds. In fact, Hadoop

ecosystem is undoubtedly one of the most complete solutions for data analysis problem, but at the same time, it is thought for high-skilled users.

On the other hand, many other solutions are designed for low-skilled users or for low-medium organizations that do not want to spend resources in developing and maintaining enterprise data analysis solutions. Two representative examples of such data analysis solutions are Microsoft Azure Machine Learning and Data Mining Cloud Framework.

Microsoft Azure Machine Learning (Azure ML) (https://azure.microsoft.com/services/machine-learning-studio/) is a SaaS for the creation of machine learning workflows. It provides a very high level of abstraction, because a programmer can easily design and execute data analytics applications by using simple drag-and-drop web interface and exploiting many built-in tools for data manipulation and machine learning algorithms.

The Data Mining Cloud Framework (DMCF) (Marozzo et al. 2015) is a software system developed at University of Calabria for allowing users to design and execute data analysis workflows on Clouds. DMCF supports a large variety of data analysis processes, including single-task applications, parameter-sweeping applications, and workflow-based applications. A workflow in DMCF can be developed using a visual- or a script-based language. The visual language, called VL4Cloud (Marozzo et al. 2016), is based on a design approach for end users having a limited knowledge of programming paradigms. The script-based language, called JS4Cloud (Marozzo et al. 2015), provides a flexible programming paradigm for skilled users who prefer to code their workflows through scripts.

Other solutions have been created mainly for scientific research purposes, and, for this reason, they are poorly used for developing business applications (e.g., E-Science Central, COMPSs, and Sector/Sphere).

e-Science Central (e-SC) (Hiden et al. 2013) is a Cloud-based system that allows scientists to store, analyze, and share data in the Cloud. It provides a user interface that allows programming visual workflows in any Web browser.

e-SC is commonly used to provide a data analysis back end to stand-alone desktop or Web applications. To this end, the e-SC API provides a set of workflow control methods and data structures. In the current implementation, all the workflow services within a single invocation of a workflow execute on the same Cloud node.

COMPSs (Lordan et al. 2014) is a programming model and an execution runtime, whose main objective is to ease the development of workflows for distributed environments, including private and public Clouds. With COMPSs, users create a sequential application and specify which methods of the application code will be executed remotely. Providing an annotated interface where these methods are declared with some metadata about them and their parameters does this selection. The runtime intercepts any call to a selected method creating a representative task and finding the data dependencies with all the previous ones that must be considered along the application run.

Sector/Sphere (Gu and Grossman 2009) is an open-source Cloud framework designed to implement data analysis applications involving large, geographically distributed datasets. The framework includes its own storage and compute services, called Sector and Sphere, respectively, which allow to manage large dataset with high performance and reliability.

Examples of Application

Cloud computing has been used in many scientific fields, such as astronomy, meteorology, social computing, and bioinformatics, which are greatly based on scientific analysis on large volume of data. In many cases, developing and configuring Cloud-based applications requires a high level of expertise, which is a common bottleneck in the adoption of such applications by scientists.

Many solutions for Big Data analysis on Clouds have been proposed in bioinformatics, such as Myrna (Langmead et al. 2010), which

is a Cloud system that exploits MapReduce for calculating differential gene expression in large RNA-seq datasets.

Wang et al. (2015) propose a heterogeneous Cloud framework exploiting MapReduce and multiple hardware execution engines on FPGA to accelerate the genome sequencing applications.

Cloud computing has been also used for executing complex Big Data mining applications. Some examples are as follows: Agapito et al. (2013) perform an association rule analysis between genome variations and clinical conditions of a large group of patients; Altomare et al. (2017) propose a Cloud-based methodology to analyze data of vehicles in a wide urban scenario for discovering patterns and rules from trajectory; Kang et al. (2012) present a library for scalable graph mining in the Cloud that allows to find patterns and anomalies in massive, real-world graphs; and Belcastro et al. (2016) propose a model for predicting flight delay according to weather conditions.

Several other works exploited Cloud computing for conducting data analysis on large amount of data gathered from social networks. Some examples are as follows:

You et al. (2014) propose a social sensing data analysis framework in Clouds for smarter cities, especially to support smart mobility applications (e.g., finding crowded areas where more transportation resources need to be allocated); Belcastro et al. (2017) present a Java library, called ParSoDA (Parallel Social Data Analytics), which can be used for developing social data analysis applications.

Future Directions for Research

Some of most important research trends and issues to be addressed in Big Data analysis and Cloud systems for managing and mining large-scale data repositories are:

- *Data-intensive computing*. The design of data-intensive computing platforms is a very significant research challenge with the goal of building computers composed of a large number of multi-core processors. From a software point of view, these new computing platforms open big issues and challenges for software tools and runtime systems that must be able to manage a high degree of parallelism and data locality. In addition, to provide efficient methods for storing, accessing, and communicating data, intelligent techniques for data analysis and scalable software architectures enabling the scalable extraction of useful information and knowledge from data are needed.
- *Massive social network analysis*. The effective analysis of social network data on a large scale requires new software tools for real-time data extraction and mining, using Cloud services and high-performance computing approaches (Martin et al. 2016). Social data streaming analysis tools represent very useful technologies to understand collective behaviors from social media data. New approaches to data exploration and model visualization are necessary taking into account the size of data and the complexity of the knowledge extracted.
- *Data quality and usability*. Big Data sets are often arranged by gathering data from several heterogeneous and often not well-known sources. This leads to a poor data quality that is a big problem for data analysts. In fact, due to the lack of a common format, inconsistent and useless data can be produced as a result of joining data from heterogeneous sources. Defining some common and widely adopted format would lead to data that are consistent with data from other sources, that means high-quality data.
- *In-memory analysis*. Most of the data analysis tools access data sources on disks, while, differently from those, in-memory analytics access data in main memory (RAM). This approach brings many benefits in terms of query speed up and faster decisions. In-memory databases are, for example, very effective in real-time data analysis, but they require high-performance hardware support and fine-grain parallel algorithms (Tan et al. 2015). New 64-bit operating systems allow to

address memory up to one terabyte, so making realistic to cache very large amount of data in RAM. This is why this research area has a strategic importance.

- *Scalable software architectures for fine-grain in-memory data access and analysis*. Exascale processors and storage devices must be exploited with fine-grain runtime models. Software solutions for handling many cores and scalable processor-to-processor communications have to be designed to exploit exascale hardware (Mavroidis et al. 2016).

References

Agapito G, Cannataro M, Guzzi PH, Marozzo F, Talia D, Trunfio P (2013) Cloud4snp: distributed analysis of SNP microarray data on the cloud. In: Proceedings of the ACM conference on bioinformatics, computational biology and biomedical informatics 2013 (ACM BCB 2013). ACM, Washington, DC, p 468. ISBN:978-1-4503-2434-2

Altomare A, Cesario E, Comito C, Marozzo F, Talia D (2017) Trajectory pattern mining for urban computing in the cloud. Trans Parallel Distrib Syst 28(2):586–599. ISSN:1045-9219

Belcastro L, Marozzo F, Talia D, Trunfio P (2016) Using scalable data mining for predicting flight delays. ACM Trans Intell Syst Technol. ACM, New York, 8(1): 5:1–5:20

Belcastro L, Marozzo F, Talia D, Trunfio P (2016, to appear) Using scalable data mining for predicting flight delays. ACM Trans Intell Syst Technol (ACM TIST)

Belcastro L, Marozzo F, Talia D, Trunfio P (2017) A parallel library for social media analytics. In: The 2017 international conference on high performance computing & simulation (HPCS 2017), Genoa, pp 683–690. ISBN:978-1-5386-3250-5

Dean J, Ghemawat S (2004) Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on symposium on operating systems design & implementation, OSDI'04, Berkeley, vol 6, pp 10–10

Gu Y, Grossman RL (2009) Sector and sphere: the design and implementation of a high-performance data cloud. Philos Trans R Soc Lond A Math Phys Eng Sci 367(1897):2429–2445

Hiden H, Woodman S, Watson P, Cala J (2013) Developing cloud applications using the e-science central platform. Philos Trans R Soc A 371(1983):20120085

Kang U, Chau DH, Faloutsos C (2012) Pegasus: mining billion-scale graphs in the cloud. In: 2012 IEEE international conference on acoustics, speech and signal processing (ICASSP), pp 5341–5344. https://doi.org/10.1109/ICASSP.2012.6289127

Langmead B, Hansen KD, Leek JT (2010) Cloud-scale rna-sequencing differential expression analysis with Myrna. Genome Biol 11(8):R83

Li A, Yang X, Kandula S, Zhang M (2010) Cloudcmp: comparing public cloud providers. In: Proceedings of the 10th ACM SIGCOMM conference on Internet measurement. ACM, pp 1–14

Lordan F, Tejedor E, Ejarque J, Rafanell R, Álvarez J, Marozzo F, Lezzi D, Sirvent R, Talia D, Badia R (2014) Servicess: an interoperable programming framework for the cloud. J Grid Comput 12(1):67–91

Marozzo F, Talia D, Trunfio P (2015) Js4cloud: script-based workflow programming for scalable data analysis on cloud platforms. Concurr Comput Pract Exp 27(17):5214–5237

Marozzo F, Talia D, Trunfio P (2016) A workflow management system for scalable data mining on clouds. IEEE Trans Serv Comput, vol PP(99), p 1

Martin A, Brito A, Fetzer C (2016) Real-time social network graph analysis using streammine3g. In: Proceedings of the 10th ACM international conference on distributed and event-based systems, DEBS'16. ACM, New York, pp 322–329

Mavroidis I, Papaefstathiou I, Lavagno L, Nikolopoulos DS, Koch D, Goodacre J, Sourdis I, Papaefstathiou V, Coppola M, Palomino M (2016) Ecoscale: reconfigurable computing and runtime system for future exascale systems. In: 2016 design, automation test in Europe conference exhibition (DATE), pp 696–701

Mell PM, Grance T (2011) Sp 800-145. The nist definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg

Richardson L, Ruby S (2008) RESTful web services. O'Reilly Media, Inc., Newton

Talia D, Trunfio P, Marozzo F (2015) Data analysis in the cloud. Elsevier. ISBN:978-0-12-802881-0

Tan KL, Cai Q, Ooi BC, Wong WF, Yao C, Zhang H (2015) In-memory databases: challenges and opportunities from software and hardware perspectives. SIGMOD Rec 44(2):35–40

Wang C, Li X, Chen P, Wang A, Zhou X, Yu H (2015) Heterogeneous cloud framework for big data genome sequencing. IEEE/ACM Trans Comput Biol Bioinform 12(1):166–178. https://doi.org/10.1109/TCBB.2014.2351800

You L, Motta G, Sacco D, Ma T (2014) Social data analysis framework in cloud and mobility analyzer for smarter cities. In: 2014 IEEE international conference on service operations and logistics, and informatics (SOLI), pp 96–101

Cloud Databases

▸ Databases as a Service

Cloud-Based SQL Solutions for Big Data

Marcin Zukowski
Snowflake Computing, San Mateo, CA, USA

Overview

Cloud computing is one of the most important trends in the current software industry. Cloud possesses unique technical and business characteristics, enabling new approaches to software design and new usage models. In this chapter we present the key characteristics of the cloud systems, and discuss opportunities and challenges traditional database systems face when deployed on this new platform. Using a set of existing systems, we demonstrate various approaches to building cloud-based SQL Big Data solutions.

Cloud

Cloud computing is possibly the largest shift in computing since the client-server model became popular. In recent years, we see companies of all sizes embrace it, often for very different reasons. Architecturally, it introduces a lot of previously unavailable features, that provide amazing opportunities to system and application developers. At the same time, careful (re)design of software is needed to take full advantage of them.

The term "Cloud" tends to be used in various contexts:

- An abstract remote location where data is stored, and processing takes place (e.g., "my photos are in the cloud");
- Infrastructure-as-a-Service (IaaS) solutions, that mostly replace on-premise physical entities like buildings, (virtual) compute and storage resources and networking infrastructure;
- Platform-as-a-Service (PaaS) solutions, extending IaaS with additional services (e.g., database systems, messaging services etc.) enabling development of end-user applications;
- Software-as-a-Service (SaaS) solutions, encapsulating the complete application functionality in an end-user system;
- Single-vendors cloud systems, e.g., Amazon Web Services (AWS). They typically provide IaaS (e.g., EC2 for AWS) and PaaS (e.g., SQS and Lambda) solutions, and sometimes also end-user SaaS solutions (e.g., AWS Quicksight). This meaning is used mostly in this article.

Historically, multiple companies made attempts to build cloud platforms. However, the technical complexity and economies of scale led to only a few companies dominating the market (Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform), with a reasonably small number of other players (Magic Quadrant for Cloud Infrastructure as a Service, Worldwide).

Cloud Platform Characteristics

Cloud systems provide a new, unique set of features not available previously, which offer exciting opportunities for database systems.

Elastic Compute

On-demand provisioning of computing resources is a defining feature of cloud platforms. It allows quickly (typically in minutes) getting access to (effectively) arbitrarily large amount of compute nodes. Additionally, these resources can be as-needed scaled up and down, resulting in users paying only for the actual usage.

This is dramatically different from traditional on-premise situation, where hardware resources:

- Had to be planned months in advance, leading to high capital expenses (CapEx) and slow project rollouts
- Were provisioned for peak usage (further increasing CapEx and reducing cost efficiency)

or average utilization (leading to insufficient performance at peak times)
- Required continuous maintenance (high operating expense – OpEx)
- Required additional infrastructure (buildings, power) impacting both CapEx and OpEx

Additional elasticity aspects of cloud compute layers include:

- Instance types, e.g., instances with more RAM VS more storage, or with specialized hardware components like GPUs or FPGAs
- Payment models, e.g., fully on-demand or upfront for a given period
- Availability models, e.g., cheaper spot instances in AWS, that can be taken away anytime, and "reserved" instances that provide a better cost efficiency for highly-utilized systems
- Ability to use various storage layers, see below

An elastic compute layer is present in all cloud systems, e.g., AWS EC2, Google Compute Engine and Azure VMs.

Storage Services

Most cloud systems provide multiple layers of storage systems, differing on a number of dimensions. The table below lists the most popular storage layers from leading providers and their vendor-specific services. These different services can lead to very different software design choices, depending on the requirements of a given system or application (Table 1).

Additional Features and Opportunities

Multi-Tenancy A unique aspect of cloud systems is that each cloud platform serves thousands of different users. On the cloud provider side, this leads to economy-of-scale benefits, savings from sharing resources, and higher predictability of general usage trends. On the user side, it gives access to a huge pool of resources. It also opens the possibility of sharing access to the same data, if its stored in a shared layer (like S3). As a result, sharing of data is possible without a physical data movement task (e.g., transferring files) and becomes to a logical operation (e.g., granting access). This allows completely new classes of applications and systems.

Geographical distribution Most cloud systems are internally distributed across multiple (typically 3) physical data centers. As a results many (but not all) services provide resiliency not only to single-machine or single-rack failures, but to the entire data center failures. This provides the

Cloud-Based SQL Solutions for Big Data, Table 1 Example storage services present in various cloud platforms

	Cost	Size	Latency	Bandwidth	Persistency	Access	AWS (Amazon EC2: Storage)	Azure (Introduction to Microsoft Azure Storage)	Google (Google Cloud Platform: Storage Options)
Local instance storage	Free (included)	Limited, fixed	Very low	Very high	Ephemeral	Single instance	Available (not all types)	Available	Available (not all types)
Distributed block storage	Medium	Limited, elastic	Low	High	Persistent	Single instance (at a time)	EBS	General purpose storage	Persistent disks
Distributed file system	High	Unlimited	Medium	Medium	Persistent	Shared	EFS	Azure files	*Not available*
Object (blob) store	Very low	Unlimited	High	Low	Persistent	Shared	S3	Azure blob storage	Cloud storage

level of availability previously only available to the largest companies.

Security Cloud systems offer improvements to system security at various levels: physical security (state of the art systems deployed at scale); network security (e.g., Virtual Private Cloud in AWS); encryption (automatic storage encryption; encryption key management); authentication and identity management (IAM in AWS); auditing and more. As a result, building secure systems in a cloud is often easier than providing a similar (externally facing) system on-premise.

Additional services Cloud vendors offer dozens of additional managed services, in areas like (examples for AWS): messaging systems (e.g., SMS, SQS), databases (e.g., RDS, DynamoDB), serverless computing (e.g., Elastic Beanstalk, Lambda), management (e.g., CloudTrail, CloudWatch) and more. These powerful tools make creating applications much easier in the cloud environment.

SaaS While IaaS provides flexibility and cost benefits to organizations, a good software-as-a-service product can lead also to a higher productivity, better user satisfaction, and personnel cost reduction. As a result, cloud systems often focus on providing a low-maintenance, mostly automated and easy-to use experience. For example, aspects like friendly user interfaces, automatic software update management, high-availability, built-in monitoring etc are all now expected by most users of cloud systems.

Cloud System Challenges

While providing a lot of great features, cloud system introduce challenges which application designers need to incorporate into their architecture. This section lists some of them.

Unpredictability of single-instance performance Since resources in the cloud are typically shared and virtualized, it is possible for one tenant of the same physical resource to have a negative impact on other users. Modern virtualization technologies prevent that to a large extent, but the problem is not completely gone.

Increased failure rates similarly, it is more common for instances in the compute layer to die without a warning, due to hardware or software corruption, often caused by activity unrelated to a given user.

Non-transparent system topology For the end-user, the mapping of virtual compute instances onto physical entities is not visible. As a result, one cant say if two nodes might influence each others behavior, or predict the cost of the network communication.

Unpredictability of services stability and performance While cloud services typically offer very high availability, some of them behave inconsistently. For example, it is known that a simple read request to AWS S3, while usually taking a fraction of a second, occasionally can take multiple seconds.

Limited hardware choice Many on premise systems use carefully tuned hardware units, with balanced CPU/RAM/disk and network. While cloud offers multiple instance types, it is often not possible to get the exact hardware configuration that would be optimal for a given application.

Cloud and Database Systems

The previously described properties of the cloud clearly demonstrate these systems are significantly different from on-premise infrastructure available to most companies.

It is perfectly possible to deploy existing software in a cloud environment, and a lot of database systems in fact are deployed this way. However, such strategy often does not allow benefiting from the unique cloud opportunities, and might not handle cloud-specific problems well.

Let us look at a list of various challenges that database system designers face when designing for the cloud

Elasticity Most popular design for on-premise large scale database systems is "shared nothing" (Stonebraker 1985). This is a great architecture for fixed-topology systems, but presents a lot of problems when highly elastic behavior is desired. As a result, other approaches have been proposed.

Scalability Combination of elastic compute resources and pay-for-use billing model, leads to customers demanding higher peak compute power (for a shorter time). As a result, distributed database systems need to be designed to efficiently scale to larger sizes.

Infrastructure reliability In traditional systems, an infrastructure failure (failed disk, network partitioning etc) and performance degradations were often consider exceptional events. In the cloud, they are more likely and systems need to be designed for them.

Performance consistency The performance differences between various nodes participating in query processing can be much higher in a shared environment. As a result, features like load balancing and skew handling are significantly more important in the cloud.

System topology With most of the cloud infrastructure virtualized, the placement of logical entities on physical hardware is often unknown. This causes problems for aspects like fault tolerance, where a single machine failure can potentially influence multiple logical instances, making it much harder to provide system failure guarantees.

Network efficiency Many high-performance distributed database systems are designed for high-performance networking interfaces like InfiniBand – when suddenly faced with e.g., a 1 gigabit Ethernet connection, their performance can degrade quickly. This influences aspects like distributed algorithms, data exchange formats etc.

Security While being typically deployed in isolated, private environments, databases did not have to worry about many aspects of security present in the cloud environments. For example, for many users, having all data encrypted in-flight is legally required in a shared environment.

Extensibility Many database systems allow using custom software to enhance various features of the system (for example, C or C++ user defined functions). Shared systems need to devise mechanisms protecting against malicious code provided through these mechanisms.

Monitoring On-premise databases could assume the users had system-level access to the instances, giving them insight into aspects like memory or disk utilization. With databases provided as a service, they often do not have this access, and additional levels of monitoring needs to be exposed via separate interfaces.

Client connectivity Databases traditionally assumed the user maintain a network connection throughout their session. However, with web user interfaces and users being more mobile, the need for clients that do not depend on continuous connection arises.

External data Traditional data warehouses could assume they managed all (or most) of the data they needed to access. With the recent data explosion, many organizations keep a lot of data in low-cost services like S3. Ability to efficiently access these becomes highly desired, especially with that data being easily accessible within a given cloud system.

Simplicity Another area where databases do not match the SaaS world perfectly is the system usability. Databases are famous for the complexity of their management, with activities like tuning,

backups, monitoring consume a lot of precious personnel time.

Pricing Traditional database license was reasonably simple, with the user typically paying for the volume of data or the amount of used hardware resources. With the clouds elasticity, new payment models had to be invented, typically focused on actual system usage.

Multi-tenancy Cloud database systems that choose to provide multi-tenant services face additional challenges. For example, workload of one user might adversely influence other users. Additionally, any layers of the system that are shared require strong access control mechanisms. Finally, multi-tenancy can cause scalability challenges for the shared system components.

Example Systems

In this section we discuss a few representative examples of database systems provided as cloud services. While they obviously differ in many database-specific areas, like SQL support, performance etc, we focus on their very different approaches to building a cloud data warehousing system.

Amazon Redshift

Amazon Redshift (Amazon Redshift) was the first widely available cloud data warehousing system and is still the market leader. It is derived from the on-premise Paraccel database, which followed the traditional shared nothing model (Chen et al. 2009). It uses user-sized EC2 clusters for both processing and storage.

While originally keeping the major design decisions unchanged, Redshift did introduce a number of cloud-focused improvements and extensions (Gupta et al. 2015), mainly:

- Ability to scale up/down and pause/resume, providing much better elasticity than in the on-premise systems
- Integration with S3, simplifying data backups
- Rich user interface for managing and monitoring

Additionally, in 2017 Amazon introduced a major architectural extension for Redshift called Spectrum (Amazon Redshift Spectrum). It is a sub-system focused on scanning data in S3, where a part of the Redshift query can be pushed to a separate system, which uses highly parallel compute infrastructure for operations like scans, filters and aggregations.

Microsoft Azure SQL Data Warehouse

Azure SQL Data Warehouse (SQLDW) (Microsoft Azure SQL Data Warehouse) is a cloud-focused evolution of the SQL Server Parallel Data Warehouse (Parallel Data Warehouse overview), which in turn extended SQL Server with shared-nothing architecture derived from DATAllegro (DATAllegro).

Azure SQLDW uses compute nodes accessing data stored in a distributed block store. All the data is up-front partitioned in the storage layer into a limited (60) set of partitions. Each compute node owns a subset of them, and inter-node data transfer is required to access data owned by other nodes. As such, the data assignment approach and processing layer of Azure SQLDW are similar to shared nothing systems. Still, using a separate storage layer provides some benefits of the shared filesystem approaches. For example redistributing the data between nodes is a logical operations (no data copying is needed) and is relatively fast.

Google BigQuery

Google BigQuery (Google BigQuery) has a very different lineage than Redshift and Azure SQLDW. It is based on internal Google products (BigQuery under the hood), mainly Dremel (Melnik et al. 2010) for data processing and Colossus distributed file system (Fikes) for data storage, following the shared-storage architecture.

Originally Dremel was a reasonably simple internal query processing system with limited functionality (e.g., no real distributed joins;

append-only), with a limited, SQL-like language. After releasing it publicly as BigQuery in 2011, Google has implemented a number of significant improvements bringing it much closer to raditional database systems, including a more-standard SQL support and update capability.

BigQuery is unique in the discussed set of products with its usage model – the user issues a query and only pays for the amount of data that query processes. This is the closest to the pure SaaS approach, with users not having to manage anything related to resource allocation. With this approach, it is also possible for a single query to use tens or even hundreds of machines, depending on what is currently available, with the users cost staying the same.

BigQuery provides a demonstration of how highly distributed compute layer combined with an efficient distributed storage can provide exceptional query performance without any up-front investment.

Snowflake Elastic Data Warehouse

Snowflake (Introducing Snowflake) is a rare example of a SQL data warehousing system designed from scratch with cloud platforms in mind. Its architecture (Dageville et al. 2016) is driven directly by various attributes of AWS:

- Object store is the cheapest and most reliable storage, leading to using S3 as the persistent data layer
- S3 has no update capability, leading to using immutable micro-partitions as data storage format
- S3 has low performance, leading to columnar storage, heavy data skipping, and local caching
- For compute layer to easily scale up and down, compute nodes should be stateless
- With data in S3, multiple compute nodes can access it at the same time, as long as someone coordinates their activity.
- S3 is not good for frequent data access and compute nodes are stateless, hence an extra management and metadata entity is needed
- In a multi-tenant system, all the barriers are purely logical, hence sharing data between users is possible

The result multi-cluster shared-data architecture uniquely translates a lot of cloud benefits to the end user, providing high elasticity, scalability and novel features.

Other Systems

While four systems discussed in this chapter are highly representative examples, there are many other products in this space, providing additional unique cloud-related approaches and features. That list includes, but is not limited to: Teradata Intellicloud (Teradata IntelliCloud), Micro-Focus Vertica (including the Eon mode) (Whats New in Vertica 9.0: Eon Mode Beta), Pivotal Greenplum (Pivotal Greenplum on Amazon Web Services), IBM DashDB (IBM dashDB), Oracle Autonomous Data Warehouse (Oracle Autonomous Data Warehouse Cloud) and Amazon Athena (Amazone Athena).

Conclusions

It seems clear today that most future data processing needs will be fulfilled by cloud systems. This chapter discussed the challenges and opportunities cloud brings, as well as approaches to addressing them. A short overview of a few example systems demonstrates that databases can take very different architectural approaches to this new platform and still build very successful cloud systems. With cloud adoption growing at a rapid pace, we expect a lot of innovation in this space in the coming years.

References

Amazon EC2: Storage. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Storage.html
Amazon Redshift. https://aws.amazon.com/redshift
Amazon Redshift Spectrum. https://aws.amazon.com/redshift/spectrum/
Amazone Athena. https://aws.amazon.com/athena

BigQuery under the hood. https://cloud.google.com/blog/big-data/2016/01/bigquery-under-the-hood
Chen Y et al (2009) Partial join order optimization in the paraccel analytic database In: Proceedings of SIGMOD
Dageville B et al (2016) The snowflake elastic data warehouse In: Proceedings of SIGMOD
DATAllegro. https://en.wikipedia.org/wiki/DATAllegro
Fikes A Storage architecture and challenges. https://cloud.google.com/files/storage_architecture_and_challenges.pdf
Google BigQuery. https://cloud.google.com/bigquery
Google Cloud Platform: Storage Options. https://cloud.google.com/compute/docs/disks/
Gupta A et al (2015) Amazon redshift and the case for simpler data warehouses In: Proceedings of SIGMOD
IBM dashDB. https://www.ibm.com/ms-en/marketplace/cloud-data-warehouse
Introducing Snowflake. https://www.snowflake.net/product/
Introduction to Microsoft Azure Storage. https://docs.microsoft.com\kern1pt/en-us\kern1pt/azure\kern1pt/storage\kern1pt/common/storage-introduction
Magic Quadrant for Cloud Infrastructure as a Service, Worldwide. https://www.gartner.com/doc/reprints?id=1-2G2O5FC&ct=150519
Melnik S, Gubarev A, Long JJ, Romer G, Shivakumar S, Tolton M, Vassilakis T (2010) Dremel: interactive analysis of web-scale datasets. In: Proceedings of VLDB
Microsoft Azure SQL Data Warehouse. https://azure.microsoft.com/en-us/services/sql-data-warehouse
Oracle Autonomous Data Warehouse Cloud. https://cloud.oracle.com/en_US/datawarehouse
Parallel Data Warehouse overview. https://docs.microsoft.com\kern1pt/en-us\kern1pt/sql\kern1pt/analytics\kern1pt-\kern1ptplatform-system/parallel-data-warehouse-overview
Pivotal Greenplum on Amazon Web Services. https://pivotal.io/partners/aws/pivotal-greenplum
Stonebraker M (1985) The case for shared nothing. In: Proceedings of HPTS
Teradata IntelliCloud. https://www.teradata.com/products-and-services/intellicloud
Whats New in Vertica 9.0: Eon Mode Beta. https://my.vertica.com/blog/whats-new-vertica-9-0-eon-mode-beta

Cloudlets

▸ Big Data and Fog Computing

Cluster Scheduling

▸ Advancements in YARN Resource Manager

Clustering of Process Instances

▸ Trace Clustering

CODAIT/Spark-Bench

▸ SparkBench

Collective Schema Matching

▸ Holistic Schema Matching

Columnar Storage Formats

Avrilia Floratou
Microsoft, Sunnyvale, CA, USA

Definitions

Row Storage: A data layout that contiguously stores the values belonging to the columns that make up the entire row.

Columnar Storage: A data layout that contiguously stores values belonging to the same column for multiple rows.

Overview

Fast analytics over Hadoop data has gained significant traction over the last few years, as multiple enterprises are using Hadoop to store data coming from various sources including operational systems, sensors and mobile devices, and web applications. Various Big Data frameworks have been developed to support fast analytics on top of this data and to provide insights in near real time.

A crucial aspect in delivering high performance in such large-scale environments is the underlying data layout. Most Big Data frameworks are designed to operate on top of data stored in various formats, and they are extensible enough to incorporate new data formats. Over the years, a plethora of open-source data formats have been designed to support the needs of various applications. These formats can be *row* or *column* oriented and can support various forms of serialization and compression. The *columnar* data formats are a popular choice for fast analytics workloads. As opposed to row-oriented storage, columnar storage can significantly reduce the amount of data fetched from disk by allowing access to only the columns that are relevant for the particular query or workload. Moreover, columnar storage combined with efficient encoding and compression techniques can drastically reduce the storage requirements without sacrificing query performance.

Column-oriented storage has been successfully incorporated in both disk-based and memory-based relational databases that target OLAP workloads (Vertica 2017). In the context of Big Data frameworks, the first works on columnar storage for data stored in HDFS (Apache Hadoop HDFS 2017) have appeared around 2011 (He et al. 2011; Floratou et al. 2011). Over the years, multiple proposals have been made to satisfy the needs of various applications and to address the increasing data volume and complexity. These discussions resulted in the creation of two popular columnar formats, namely, the Parquet (Apache Parquet 2017) and ORC (Apache ORC 2017) file formats. These formats are both open-source and are currently supported by multiple proprietary and open-source Big Data frameworks. Apart from columnar organization, the formats provide efficient encoding and compression techniques and incorporate various statistics that enable predicate pushdown which can further improve the performance of analytics workloads.

In this article, we first present the major works in disk-based columnar storage in the context of Big Data systems and Hadoop data. We then provide a detailed description of the Parquet and ORC file formats which are the most widely adopted columnar formats in current Big Data frameworks. We conclude the article by highlighting the similarities and differences of these two formats.

C

Related Work

The first works on columnar storage in the context of Hadoop, namely, the RCFile (Row Columnar File) (He et al. 2011) and the CIF (Column Input File format) (Floratou et al. 2011) layouts, were mostly targeting the MapReduce framework (Dean and Ghemawat 2008). These two columnar formats have adopted significantly different designs. The CIF file format stored each column in the data as a separate file, whereas the RCFile adopted a PAX-like (Ailamaki et al. 2001) data layout where the columns are stored next to each other in the same file.

These design choices led to different tradeoffs. The CIF file format was able to provide better performance as it allowed accessing only the files that contain the desired columns but required extra logic in order to colocate the files that contain adjacent columns. Without providing colocation in HDFS, reconstructing a record from multiple files would result in high network I/O. The RCFile, on the other hand, did not require any such logic to be implemented as all the columns were stored in the same HDFS block. However, the RCFile layout made it difficult to enable efficient skipping of columns due to file system prefetching. As a result, queries that were accessing a small number of columns would pay a performance overhead (Floratou et al. 2011).

The successor of the RCFile format, namely, the ORC (Optimized Row Columnar) file format, was developed to overcome the limitations of the RCFile format. It was originally designed to speed up Hadoop and Hive, but it is currently incorporated in many other frameworks as well. The ORC file format still uses a PAX-like (Ailamaki et al. 2001) layout but can more efficiently skip columns by organizing the data in larger blocks called row groups. A detailed

description of the ORC file format is provided in the following section.

The Parquet (Apache Parquet 2017) file format is another popular, open-source columnar format. The Parquet file format was designed to efficiently support queries over deeply nested data. The format is based on the Dremel distributed system (Melnik et al. 2010) that was developed at Google for interactively querying large datasets. Similar to the ORC format, the parquet format has also adopted a PAX-like layout (Ailamaki et al. 2001). The following section presents Parquet's layout in detail.

Detailed comparisons of various columnar formats can be found in Huai et al. (2013) and in Floratou et al. (2014). The work by Floratou et al. (2014) compares the ORC and Parquet file formats in the context of Hive (Apache Hive 2017) and Impala (Kornacker et al. 2015) for SQL workloads.

Other related open-source technologies are Apache Arrow (2017) and Apache Kudu (2017). Apache Arrow is a columnar in-memory format that can be used on top of various disk-based storage systems and file formats to provide fast, in-memory analytical processing. Note that as opposed to the file formats discussed above, Arrow is an in-memory representation and not a disk-based file format. Apache Kudu is an open-source storage engine that complements HDFS (Apache Hadoop HDFS 2017) and HBase (Apache Hbase 2017). It provides efficient columnar scans as well as inserts and updates. Kudu provides mutable storage, whereas the file formats described above (e.g., Parquet) are immutable, and thus any data modifications require rewriting the dataset in HDFS.

Columnar File Formats for Big Data Systems

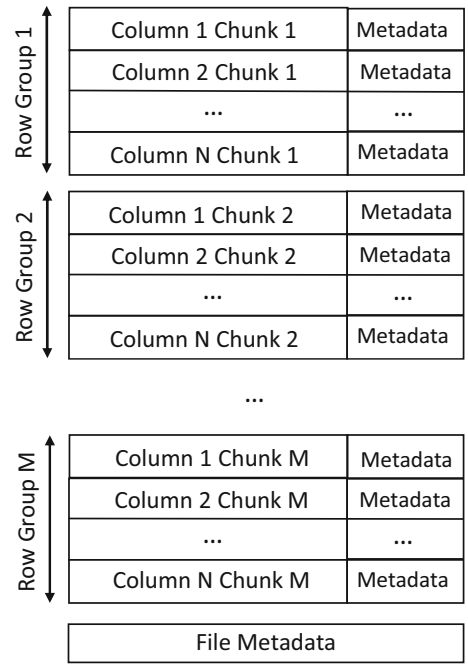
In this section, we describe in more detail two popular, open-source columnar formats that have been incorporated in various Big Data frameworks, namely, the **ORC** and the **Parquet** file formats. Both ORC and Parquet are Apache projects (Apache ORC 2017; Apache Parquet 2017) and are actively used by multiple organizations when performing analytics over Hadoop data.

The Parquet File Format

The Parquet file format (Apache Parquet 2017) is a self-described columnar data format specifically designed for the Hadoop ecosystem. It is supported by many Big Data frameworks such as Apache Spark (Zaharia et al. 2012) and Apache Impala (Kornacker et al. 2015), among others. The format inherently supports complex nested data types as well as efficient compression and encoding schemes. In the next section, we describe the format layout in more detail.

File Organization

The Parquet format supports both primitive (e.g., integer, Boolean, double, etc.) and complex data types (maps, lists, etc.). The structure of a Parquet file is shown in Fig. 1. The file consists of a set of `row groups` which essentially represent horizontal partitions of the data. A `row group` consists of a set of `column chunks`. Each



Columnar Storage Formats, Fig. 1 The parquet file format

column chunk contains data from a particular data column and it is guaranteed to be contiguous in the file. The column chunk consists of one or more pages which are the unit of compression and encoding. Each page contains a header that describes the compression and encoding method used for the data in the page.

As shown in Fig. 1, the file consists of N columns spread across M row groups. The row groups are typically large (e.g., 1 GB) to allow for large sequential I/Os. Since an entire row group might need to be accessed, the HDFS block size should be large enough to fit the row group. A typical configuration is to have 1 GB HDFS blocks that contain one row group of 1 GB.

Metadata is stored at all the levels in the hierarchy, i.e., file, column chunk, and page. The bottom of the file contains the file metadata which include information about the data schema as well as information about the column chunks in the file. The metadata contains statistics about the data such as the minimum and maximum values in a column chunk or page. Using these statistics, the Parquet file format supports predicate pushdown which allows skipping of pages and reduces the amount of data that needs to be decompressed and parsed. The Parquet readers first fetch the metadata to filter out the column chunks that must be accessed for a particular query and then read each column chunk sequentially.

Compression

As described in the previous section, the unit of compression in the Parquet file format is the page. The format supports common compression codecs such as GZIP and Snappy (Snappy Compression 2017). Moreover, various types of encoding for both simple and nested data types are also supported. The interested reader can find more information about the various encoding techniques in Parquet Encodings (2017).

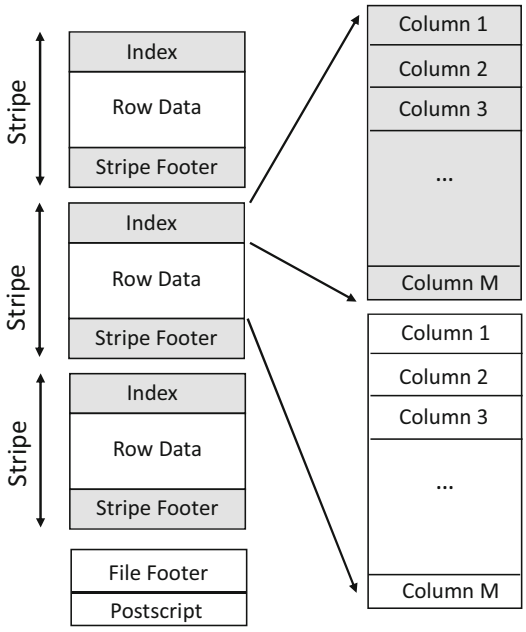
The ORC File Format

The Optimized Row Columnar (ORC) format (Apache ORC 2017) is a column-oriented storage layout that was created as part of an initiative to speed up Apache Hive (2017) queries and reduce the storage requirements of data stored in Apache Hadoop (2017). Currently the ORC file format is supported by many Big Data solutions including Apache Hive (2017), Apache Pig (2017), and Apache Spark (Zaharia et al. 2012), among others.

File Organization

The ORC format is an optimized version of the previously used Row Columnar (RC) file format (He et al. 2011). The format is self-describing as it includes the schema and encoding information for all the data in the file. Thus, no external metadata is required in order to interpret the data in the file. The format supports both primitive (e.g., integer, Boolean, etc.) and complex data types (maps, lists, structs, unions). Apart from the columnar organization, the ORC file supports various compression techniques and lightweight indexes.

The structure of an ORC file is shown in Fig. 2. As shown in the figure, the file consists of three major parts: a set of stripes, a file footer, and a postscript. The



Columnar Storage Formats, Fig. 2 The ORC file format

postscript part provides all the necessary information to parse the rest of the file such as the compression codec used and the length of the file footer. The file footer contains information related to the data stored in the file, such as the number of rows in the file, statistics about the columns, and the data schema. The process of reading the file starts by first reading the tail of the file that consists of the postscript and the file footer that contain metadata about the main body of the file.

An ORC file stores multiple groups of row data as stripes. Each stripe has a size of about 250 MB and contains only entire rows so a row cannot span multiple stripes. Internally, each stripe is divided into index data, row data, and stripe footer in that order. The data columns are stored next to each other in a PAX-like (Ailamaki et al. 2001) organization. Depending on the query's access pattern, only the necessary columns are decompressed and deserialized. The stripe footer contains the location of the columns in data as well as information about the encoding of each column. The indexes contain statistics about each column in a row group (set of 10,000 rows). For example, in the case of integer columns, the column statistics include the minimum, maximum, and sum values of the column in the given row group. These statistics can be used to skip entire sets of rows that do not satisfy the query predicates. Recently, bloom filters can additionally be used to better prune row groups (ORC Index 2017).

Finally, it is worth noting that column statistics are also stored in the file footer at the granularity of stripes. These statistics enable skipping entire stripes based on a filtering predicate.

Compression

Depending on the user's configuration, an ORC file can be compressed using a generic compression codec (typically zlib (ZLIB Compression 2017) or snappy (Snappy Compression 2017)). The file is compressed in chunks so that entire chunks can be directly skipped without decompressing the data. The default compression chunk size is 256 KB but it is configurable. Having larger chunks typically results in better compression ratios but requires more memory to be allocated during decompression.

The ORC file makes use of various run-length encoding techniques to further improve compression. The interested reader can find more information about the supported encoding methods in ORC Encodings (2017)

File Format Comparison

The Parquet and ORC file formats have both been created with the goal of providing fast analytics in the Hadoop ecosystem. Both file formats are columnar and they have adopted a PAX-like layout. Instead of storing each column at a separate file, the columns are stored next to each other in the same HDFS block. By having large row groups and HDFS blocks, both formats exploit large sequential I/Os and can skip unnecessary columns.

Both formats support various compression codecs at a fine granularity so that the amount of data that is decompressed is minimized. They also support various encoding schemes depending on the data type.

Finally, the file formats incorporate statistics at various levels in order to avoid decompressing and deserializing data that do not satisfy the filtering predicates. Typically, the statistics include the maximum and the minimum values of a column. The ORC file additionally supports bloom filters at the row group level, while the Parquet file supports statistics at the page level as well.

Regarding the data types supported, the Parquet file format has been designed to inherently support deeply nested data using the record shredding and assembly algorithm presented in Melnik et al. (2010). The ORC file, on the other hand, flattens the nested data and creates separate columns for each underlying primitive data type.

Conclusions

The increased interest in fast analytics over Hadoop data fueled the development of multiple open-source data file formats. In this work, we

focused on the columnar storage formats that are used to store HDFS data. In particular, we first reviewed the work on disk-based columnar formats for Big Data systems and then presented a detailed description of the open-source ORC and Parquet file formats. These two columnar data formats are currently supported by a plethora of Big Data solutions. Finally, we highlighted the differences and similarities of the two file formats.

Cross-References

- ▸ Caching for SQL-on-Hadoop
- ▸ Wildfire: HTAP for Big Data

References

Ailamaki A, DeWitt DJ, Hill MD, Skounakis M (2001) Weaving relations for cache performance. In: Proceedings of the 27th international conference on very large data bases (VLDB'01), pp 169–180

Apache Arrow (2017) Apache Arrow. https://arrow.apache.org/

Apache Hadoop (2017) Apache Hadoop. http://hadoop.apache.org

Apache Hadoop HDFS (2017) Apache Hadoop HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

Apache Hbase (2017) Apache HBase. https://hbase.apache.org/

Apache Hive (2017) Apache Hive. https://hive.apache.org/

Apache Kudu (2017) Apache Kudu. https://kudu.apache.org/

Apache ORC (2017) Apache ORC. https://orc.apache.org/

Apache Parquet (2017) Apache Parquet. https://parquet.apache.org/

Apache Pig (2017) Apache Pig. https://pig.apache.org/

Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51(1):107–113

Floratou A, Patel JM, Shekita EJ, Tata S (2011) Column-oriented Storage Techniques for MapReduce. Proc VLDB Endow 4(7):419–429

Floratou A, Minhas UF, Özcan F (2014) SQL-on-Hadoop: full circle back to shared-nothing database architectures. Proc VLDB Endow 7(12):1295–1306

He Y, Lee R, Huai Y, Shao Z, Jain N, Zhang X, Xu Z (2011) RCFile: a fast and space-efficient data placement structure in MapReduce-based warehouse systems. In: Proceedings of the 2011 IEEE 27th international conference on data engineering (ICDE'11). IEEE Computer Society, pp 1199–1208

Huai Y, Ma S, Lee R, O'Malley O, Zhang X (2013) Understanding insights into the basic structure and essential issues of table placement methods in clusters. Proc VLDB Endow 6(14):1750–1761

Kornacker M, Behm A, Bittorf V, Bobrovytsky T, Ching C, Choi A, Erickson J, Grund M, Hecht D, Jacobs M, Joshi I, Kuff L, Kumar D, Leblang A, Li N, Pandis I, Robinson H, Rorke D, Rus S, Russell J, Tsirogiannis D, Wanderman-Milne S, Yoder M (2015) Impala: a modern, open-source SQL engine for hadoop. In: CIDR

Melnik S, Gubarev A, Long JJ, Romer G, Shivakumar S, Tolton M, Vassilakis T (2010) Dremel: interactive analysis of web-scale datasets. Proc VLDB Endow 3(1–2):330–339

ORC Encodings (2017) ORC Encodings. https://orc.apache.org/docs/run-length.html

ORC Index (2017) ORC Index. https://orc.apache.org/docs/spec-index.html

Parquet Encodings (2017) Parquet Encodings. https://github.com/apache/parquet-format/blob/master/Encodings.md

Snappy Compression (2017) Snappy Compression. https://en.wikipedia.org/wiki/Snappy_(compression)

Vertica (2017) Vertica. https://www.vertica.com/

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, USENIX

ZLIB Compression (2017) ZLIB Compression. https://en.wikipedia.org/wiki/Zlib

Complex Event Processing

- ▸ Pattern Recognition

Component Benchmark

Klaus-Dieter Lange and David L. Schmidt
Hewlett Packard Enterprise, Houston, TX, USA

Synonyms

Microbenchmark; Worklet

Overview

Component benchmarks are valuable tools for isolating and analyzing the performance characteristics of specific subsystems within a server

environment. This chapter will provide historical context for component benchmarks, brief descriptions of the most commonly used benchmarks, and a discussion on their uses in the field.

Definitions

Software that utilizes standard applications, or their core routines that focus on a particular access pattern, in order to measure the performance and/or efficiency of one of the primary server and storage components (CPU transactions, memory, and storage/network IO).

Historical Background

From earliest days of digital computer systems, the goal of quantifying, improving, and optimizing computational performance has been a subject great interest. The earliest measures of performance were comparisons of low-level instruction execution times. A mix of several of these execution times would be combined to produce an overall rating that could be compared between systems. The most well-known of these early benchmarks was the Gibson mix, devised by Jack Gibson of IBM (Gibson 1970). As high-level computer languages were developed, more complex applications were created to develop more rigorous methods of measuring a system's performance. Whetstone (Longbottom 2014) and Dhrystone (Weiss 2002) are both early examples of high-level language benchmarks.

As computer systems became more complex, benchmarks were developed to measure the performance of the separate components of the system, such as memory, floating-point arithmetic coprocessors, and data IO. Most of these early benchmarks utilized synthetic workloads and were usually provided to users as source code that needed to be compiled on the system of interest. This allowed such benchmarks to be used on multiple platforms, but there was generally no set standard on how to compile such benchmarks or how to compare results between different architectures which limited the scope of their use. Industry-standard consortia such as Transaction Processing Performance Council (TPC, http://www.tpc.org) and Standard Performance Evaluation Corporation (SPEC, https://www.spec.org) were established in the mid-1980s in an effort to provide fair standards for measuring system-level and component-level performance of differing platforms. Several of the benchmarks developed by these consortia are intended to measure component-level performance.

Foundations

Component benchmarks are useful tools to analyze IT equipment and troubleshoot performance bottlenecks. They utilize standard applications, or their core routines that focus on a particular access pattern, in order to measure the performance of one of the primary server and storage components such as CPU transactions, memory access, storage IO, or network IO. By limiting their scope, component benchmarks achieve a deeper analysis of the targeted subsystem. At the same time component benchmarks are often easier to develop than benchmarks that stress the complete server environment and they are typically less expensive to run, because they require fewer equipment and engineering resources.

There can be an overlap in what is considered a component benchmark vs. a microbenchmark. A microbenchmark is typically more limited in scope than a component benchmark, focusing on the performance of a single feature of a component rather than the entire component. Microbenchmarks utilize synthetic workloads rather than actual user applications to measure the performance of their intended resource. Microbenchmarks are further described in a later chapter.

Following is a description of the most common component benchmarks categorized by their target subsystem in a Big Data environment.

Processor subsystem

SPEC CPU2017 (https://www.spec.org/cpu2017) is a benchmark package developed by the Standard Performance Evaluation Corporation (SPEC) to provide a comparative measure of compute-intensive performance across the widest practical range of hardware platforms. The benchmark uses workloads developed from programs from a variety of application areas, including artificial intelligence, molecular dynamics, physics, weather-forecasting, imaging, modeling, and computer development. Suites of these workloads are run to measure floating-point (SPECrate2017 Floating Point, SPECspeed2017 Floating Point) and integer (SPECrate2017 Integer, SPECspeed2017 Integer) performance. The suites are further categorized as SPECrate and SPECspeed suites. The SPECrate suites run multiple concurrent copies of each benchmark workload to provide a measure of throughput performance. The SPECspeed suites run a single copy of each benchmark workload to measure processing speed and parallelization performance.

SPEC CPU2006 (https://www.spec.org/cpu2006) is the predecessor to SPEC CPU2017 and likewise measures CPU and memory performance utilizing integer and floating-point intensive benchmark suites. There are both rate (SPECint_rate2006, SPECfp_rate2006) and speed (SPECint2006, SPECfp2006) suites to analyze different performance characteristics of the CPU and memory subsystems.

HEP-SPEC06 (Michelotto et al. 2010; HEPiX benchmarking working group) is a benchmark based on the SPEC CPU2006 benchmark suites developed by the HEPiX forum to measure processing performance of applications in a high energy physic environment. HEP-SPEC06 requires the use of the SPEC CPU2006 benchmark tools, but only runs a subset of the component workloads; only the six benchmark modules written in C++ are utilized and are compiled as 32-bit executables. The HEP-SPEC06 run script will then invoke multiple instances of the SPEC CPU2006 run script, each running a single copy of the benchmark workloads. The output results of each SPEC CPU2006 instance are then used to generate a final HEP-SPEC06 score.

Whetstone (Longbottom) was one of the first benchmarks to attempt to standardize the measurement of floating-point performance of CPUs. Developed in 1972, the benchmark workload was a representation of a set of 124 simple Whetstone ALGOL 60 compiler instructions translated into FORTRAN. The benchmark result was measured in thousands of Whetstone instructions per second (kWips), and later in millions of Whetstone instructions per second (MWIPS). The benchmark was updated over the years and ported to C, C++, Basic, and Java. It is frequently included in component benchmark suites.

Dhrystone (Weiss 2002) is a synthetic benchmark program developed in 1984 to measure integer-based processing performance. Developed utilizing metadata from several applications written in different programming languages, Dhrystone was designed to measure system program performance. For many years Dhrystone was considered the representative benchmark for general processor performance. Originally written in Ada, it was translated into C and UNIX and is still often used in component benchmark suites. The name Dhrystone is a pun on the benchmark Whetstone, another benchmark popular at the time of its development.

LINPACK (The LINPACK benchmark programs and reports; Dongarra et al. 2002) is a software library for performing numerical linear algebra on digital com-

puters. The LINPACK benchmark uses the LINPACK software library to solve a set of predetermined set of n by n linear algebra equations in order to measure a system's floating-point computing performance, measured in floating point operations per second (FLOPS). The benchmark workload is almost exclusively floating-point based, so is an excellent stressor of the processors' math and vector instruction sets. There have been several versions of LINPACK benchmarks since their creation in 1979, which include LINPACK 100 ($n = 100$), LINPACK 1000 ($n = 1000$), and HPLinpack (a highly parallelized version that can run across multiple systems). HPL is a portable implementation of the HPLinpack benchmark written in C. Precompiled LINPACK and HPL are available for certain system architectures.

Memory

STREAM (https://www.cs.virginia.edu/stream; McCalpin 1995) is a simple synthetic benchmark that measures sustainable memory bandwidth, reported in MB/s. The benchmark is specifically designed to work with dataset much larger than the available processor cache on any given system so that the results are more indicative of the performance of a very large, vector-style application. The benchmark yields four metrics, representing different memory operations: Copy, Add, Scale, and Triad. The benchmark had C and FORTRAN versions available which can run either in a single-threaded or distributed fashion. A precompiled version of this benchmark is often included in component benchmark suites.

Storage IO

SPEC SFS2014 (https://www.spec.org/sfs2014) is a benchmark suite which measures file server throughput and response time. The SPEC SFS2014 SP2 benchmark includes five workloads which measure storage performance in different application areas: database operations, software builds, video data acquisition (VDA), virtual desktop infrastructure (VDI), and electronic design automation (EDA). Each workload is independent of the others and reports the throughput (in MB/s) and overall response time (in msec) for a given number of workload instances. Netmist is the load generator, allowing the benchmark suite can be run over any version of NFS, SMB/CIFS, object-oriented, local, or other POSIX-compatible file system.

Iometer (http://www.iometer.org) is a tool which measures the performance and characterization of I/O components for single and clustered systems. It is based on a client-server model where a single client controls one or manager driver which generates I/O using one or more worker threads. Iometer performs asynchronous I/O and can access files or block devices. The tool is very flexible and allows the user to adjust the workload by configuring target disk parameters (e.g., disk size, starting sector, number of outstanding I/Os) and the workload access specifications (e.g., transfer request size, percent random/sequential distribution, percent read/write distribution, aligned I/Os, reply size, TCP/IP status, burstiness). All of the output data is collected into a CSV file for easy manipulation. One of the more commonly utilized output parameter is I/O operations per second (IOPS).

IOzone (http://www.iozone.org/docs/IOzone_msword_98.pdf) is a file system benchmark utility which generates and measures a variety of file operations. These file operations include read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants, aio_read, aio_write, and mmap. The results can be exported into useful graphs which show performance characteristics and bottlenecks of the disk I/O subsystem. The benchmark is written in C and can be run under many operating systems.

Network IO

Netperf (https://github.com/HewlettPackard/netperf) is a software application that provides network bandwidth testing between two hosts on a network. It measures performance of bulk data transfer and request/response network traffic using either TCP or UDP via BSD sockets. There are optional tests that measure performance of DLPI, UnixDomainSockets, the Fore ATM API, and the HP HiPPI LLa interface. Originally developed by Hewlett Packard, it is now available from github. Netperf has been ported to run on numerous distributions of UNIX and Linux, Windows, and VMware.

Component benchmarks are often bundled together into a suite of benchmarks that may or may not also include microbenchmarks to provide a tool that can measure several system features in a single package. Note that these suites are frequently designed for the personal computer or workstation market rather than for the server environment.

Following is a description of the more common component benchmark suites.

SERT (https://www.spec.org/sert) is the Server Efficiency Rating Tool developed by SPEC to evaluate the energy efficiency of servers. The SERT suite is a suite of 12 "worklet" modules that stress different system components and measure performance as well as the power consumption of the system. These worklets stress the CPU, memory, and disk I/O subsystem at various target load levels well, as well as one hybrid worklet that stress a series of complex CPU and memory access pattern. The SERT suite utilizes the Chauffeur (https://www.spec.org/chauffeur-wdk) harness and is configurable so individual worklets or categories of worklets can be run.

LMBench (http://lmbench.sourceforge.net) is a suite of simple benchmarks that measure bandwidth and latency performance for a variety of system components, including memory, caches, disk I/O, and network I/O. LMBench is written in C and can be used on most distributions of UNIX and Linux.

SiSoftware Sandra (http://www.sisoftware.eu) is suite of modules that provide information about a system's hardware and software, including performance measurements on the CPU, memory, disk I/O, and graphics. Only available on Windows, the primary target of the suite is PC and workstation systems. There are many versions available from the free Lite version to the Enterprise version.

3DMark (https://www.3dmark.com) is a benchmarking tool that measures performance of CPU and graphics utilizing modules based off different versions of DirectX. Only available on Windows, Android, and iOS, 3DMark is intended for analyzing PCs used by gaming enthusiasts.

PCMark (https://www.futuremark.com/benchmarks/pcmark) is a benchmarking tool similar to 3DMark which measures the performance of different components of PCs, such as CPU, memory, disk I/O, and graphics. Only available on Windows, PCMark is intended for home use.

Key Applications

Single component benchmarks generally are not directly useful for the analysis of Big Data environments, as they measure only one or a few specific aspects of one primary server or storage components. A suite of component benchmarks like the SERT suite can be a key method to measure the behaviors of the primary server and storage components. Nonetheless, they are not measuring the interaction and possible performance bottlenecks between those components. The real usefulness of component benchmarks is their role in the simplification of the performance analysis and the pinpointing of the actual performance bottlenecks within a component.

The major key application of component benchmarks is their usage during server development and the design/deployment phases of new Big Data environments. In general, the first step is to run one of the component suites (or

a series of hand-picked component benchmarks) in order to determine if one subsystem exhibits lower-than-expected performance. Next, an analysis of the subsystem is conducted to further pinpoint the root causes. For example, the performance results from the SERT suite determine lower-than-expected CPU subsystem performance. The next step would include running the SPEC CPU2017 benchmark suite to further close in on the root cause. Finally, after a possible resolution of the issues is implemented, the original component suite should be run again in order to verify that the resolution satisfactorily fixed the bottleneck. Please note that it is the nature of performance bottlenecks that they switch from one component to another, once the root cause of the original bottleneck is resolved.

In order to maintain their usefulness, component benchmarks need a continuous development cycle to keep up with emerging technologies. For example, in order to accurately measure the performance gain of a new microarchitecture, a compiled version of the benchmark code (including its libraries if applicable), which supports this new microarchitecture, should be utilized. A 0.5–3.0% performance gain from such a microarchitecture optimization is quite common.

Cross-References

- Analytics Benchmarks
- Benchmark Harness
- Business Process Performance Measurement
- End-to-End Benchmark
- Energy Benchmarking
- Energy Efficiency in Big Data Analysis
- Graph Benchmarking
- Machine Learning Benchmarks
- Metrics for Big Data Benchmarks
- Microbenchmark
- Performance Evaluation of Big Data Analysis
- SparkBench
- Stream Benchmarks
- Virtualized Big Data Benchmarks
- YCSB

References

3DMark benchmark. https://www.3dmark.com

Chauffeur harness. https://www.spec.org/chauffeur-wdk

Dongarra JJ, Luszczek P, Petitet A (2002) The LINPACK benchmark: past, present, and future. http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl paper.pdf

Gibson JC (1970) The Gibson mix. Technical report TR 00.2043. IBM Systems Development Division, Poughkeepsie

HEPiX benchmarking working group. https://www.hepix.org/e10227/e10327/e10325

Iometer project. http://www.iometer.org

IOzone filesystem benchmark. http://www.iozone.org/docs/IOzone_msword_98.pdf

LMBench tool. http://lmbench.sourceforge.net

Longbottom (2014) R. Whetstone benchmark history and results. http://www.roylongbottom.org.uk/whetstone.htm

McCalpin JD (1995) Memory bandwidth and machine balance in current high performance computers. IEEE Comput Soc Tech Comm Comput Arch (TCCA) Newsl. https://www.researchgate.net/publication/213876927_Memory_Bandwidth_and_Machine_Balance_in_Current_High_Performance_Computers

Michelotto M, Alef M, Iribarren A, Meinhard H, Wegner P, Bly M, Benelli G, Brasolin F, Degaudenzi H, De Salvo A, Gable I, Hirstius A, Hristov P (2010) A comparison of HEP code with SPEC benchmark on multi-core worker nodes. http://www.pd.infn.it/hepmark/HS06.pdf

Netperf benchmark. https://github.com/HewlettPackard/netperf

PCMark benchmark. https://www.futuremark.com/benchmarks/pcmark

Server Efficiency Rating Tool (SERT). https://www.spec.org/sert

SiSoftware Sandra. http://www.sisoftware.eu

SPEC CPU 2006 benchmark. https://www.spec.org/cpu2006

SPEC CPU 2017 benchmark. https://www.spec.org/cpu2017

SPEC SFS 2014 benchmark. https://www.spec.org/sfs2014

Standard Performance Evaluation Corporation (SPEC). https://www.spec.org

STREAM benchmark. https://www.cs.virginia.edu/stream

The LINPACK benchmark programs and reports. http://www.netlib.org/benchmark/index.html

Transaction Processing Performance Council (TPC). http://www.tpc.org

Weiss A (2002) Dhrystone benchmark: history, analysis, scores and recommendations. http://www.johnloomis.org/NiosII/dhrystone/ECLDhrystoneWhitePaper.pdf

Compressed Indexes for Repetitive Textual Datasets

Travis Gagie[1] and Gonzalo Navarro[2]
[1]EIT, Diego Portales University, Santiago, Chile
[2]Department of Computer Science, University of Chile, Santiago, Chile

Definitions

Given a text or collection of texts containing long repeated substrings, we are asked to build an index that takes space bounded in terms of the size of a well-compressed encoding of the text or texts and that, given an arbitrary pattern, can quickly report the occurrences of that pattern in the dataset.

Overview

Humanity now stores as much data in a year as we did in our whole history until the turn of the millennium. Most applications that use this data need to query it efficiently, and one of the most important kinds of queries is pattern matching in texts. It is usually impractical to scan massive textual datasets every time we want to count or find the occurrences of a pattern, so we must index them. Until fairly recently, indexing a text often took much more memory than simply storing the dataset. In 2000, however, Ferragina and Manzini (2000, 2005) showed how to simultaneously compress and index a text, with the index itself supporting access to the text and thus replacing it. Also in 2000, Grossi and Vitter (2000, 2005) and Sadakane (2000, 2003) proposed an alternative data structure with essentially the same query functionality but with a space bound proportional to the size of the uncompressed text; it was then improved to occupy compressed space. Ferragina and Manzini's and Grossi and Vitter's data structures, the FM-index and compressed suffix array (CSA), have had a dramatic impact on several fields such as bioinformatics, where FM-indexes are central to many important DNA aligners such as Bowtie (Langmead et al. 2009) and BWA (Li and Durbin 2009). We note that, in contrast to inverted lists, CSAs and FM-indexes can index any kind of text and allow any pattern to be sought, without regard for word boundaries. This is important not only for applications involving DNA but also to index source code repositories, multimedia sequences, and even tokenized word sequences in order to perform phrase searches.

CSAs and FM-indexes use space comparable to the sizes of encodings produced by the best statistical compressors, which achieve bounds in terms of higher-order entropies, but massive texts are often repetitive and are thus much more compressible using dictionary-based compressors such as LZ77 (Ziv and Lempel 1977). For example, humans are genetically almost identical, so a good dictionary-based compressor can compress a database of a thousand human genomes to about 1% of its size, as reported by the 1000 Genomes Project (http://www.internationalgenome.org). Naturally, researchers have tried to adapt CSAs and FM-indexes to take better advantage of repetitions or to find alternatives to them that do so. Their efforts can be broadly classified into four groups: run-length compressed FM-indexes, indexes based on Lempel-Ziv compression or context-free grammars, compressed directed acyclic word graphs (CDAWGs), and, most recently, graph-based indexes. We briefly survey the history of each approach, mentioning its strengths and weaknesses and giving pointers to practical implementations where they are available.

Key Research Findings

Run-Length Compressed CSAs and FM-Indexes

Consider a repetitive text (or the concatenation of the texts in a repetitive collection) $T[1..n]$. The suffix array of T, $A[1..n]$, points to the starting

positions of all the suffixes of T in lexicographic order. Assume a long string $S[1..k]$ appears s times in T. Then the s suffixes starting with $S[1..k]$ appear in a contiguous range $A[p_1..p_1 + s - 1]$. It is likely that the suffixes starting with $S[2..k]$ also appear together in another range $A[p_2..p_2 + s - 2]$, in the same order as those in $A[p_1..p_1 + s - 1]$. The same is likely to occur for any $S[k'..k]$ as long as $k - k'$ is sufficiently large. Such a regularity shows up in the Ψ function of CSAs, $\Psi[i] = A^{-1}[A[i] + 1]$, since the values in $\Psi[p_1..p_1 + s - 1]$ will point to $[p_2..p_2 + s - 2]$, and so on, thereby inducing runs of 1s in $\Psi'[i] = \Psi[i] - \Psi[i - 1]$, which is the main component of CSAs. It also shows up in the Burrows-Wheeler Transform (BWT) of T, $\mathrm{BWT}[i] = T[A[i] - 1]$, which is the main component of the FM-index: $\mathrm{BWT}[p_2..p_2 + s - 1]$ will be a run of copies of $S[1]$ and so on.

Mäkinen et al. (2010) took the first step toward adapting CSAs and FM-indexes to handle repetitive texts better, by run-length compressing the runs of 1s in Ψ' or the runs of the same letters in the BWT. The number of runs in Ψ' is almost the same as the number r of runs in the BWT. Their run-length compressed CSA and FM-index then use $O(r)$ space, often much less than a standard CSA or FM-index while still counting patterns' occurrences quickly.

Until very recently it was not known how to compress similarly the suffix-array sample with which we can locate patterns' occurrences in T, without making those locating queries extremely slow. However, Gagie et al. (2018) have now introduced a new sampling scheme that takes $O(r)$ space and lets us locate each occurrence with essentially only a predecessor query. Specifically, they can count the number of occurrences of a pattern of length m in $O(m \log \log n)$ time and then locate each occurrence in $O(\log \log n)$ time.

Lempel-Ziv and Grammar-Based Indexes

Kärkkäinen and Ukkonen (1996) showed how to store an $O(z)$-space data structure in addition to the text T, where z is the number of phrases in the LZ77 parse of T, such that locating all the occurrences of a pattern of length m takes $O(m^2 + m \log z + \sqrt{mz \log m})$ time plus $O(\log z)$ time per occurrence. They divided the occurrences into two types, called primary and secondary: the former start in one phrase and end in another (or at the next phrase boundary), and the latter are completely contained within single phrases. Their idea was to store a pair of Patricia trees, one for the reversed phrases in the parse and the other for the suffixes starting at phrase boundaries. By dividing a pattern every possible way into a nonempty prefix and a (possibly empty) suffix and searching for the reversed prefix in the first tree and the suffix in the second, they obtain a pair of Patricia tree nodes or, equivalently, a pair of ranges in the lexicographically sorted reversed prefixes and suffixes. Using an $O(z)$-space data structure for four-sided two-dimensional range reporting, they obtain all the primary occurrences. They use access to the text to verify the occurrences. Using another $O(z)$-space data structure that represents the structure of the LZ77 parse, they can then obtain all the secondary occurrences from the primary ones.

Researchers are still following Kärkkäinen and Ukkonen's basic design. The major alterations have aimed at a compressed representation of the text, which is essential in the repetitive scenario. Compressed representations, which must offer direct access in order to support searches, have been built on the LZ77 parse or a variant of it (Kreft and Navarro 2013; Do et al. 2014; Navarro 2017), a context-free grammar (Claude and Navarro 2011, 2012; Gagie et al. 2012, 2014; Bille et al. 2017), a run-length compressed FM-index without samples (which also replaces the Patricia trees) (Belazzougui et al. 2015, 2017), and CDAWGs (Belazzougui et al. 2015, 2017). Another improvement has been to substitute z-fast tries for the Patricia trees, which allows removing the quadratic dependence on m in the query time (Gagie et al. 2014; Bille et al. 2017).

There are also indexes based on edit-sensitive parsing (Maruyama et al. 2013; Takabatake et al. 2014, 2016) and locally consistent parsing (Nishimoto et al. 2016), although the former has poor worst-case query time and seems practical only for long patterns, and the latter is complicated and competitive in theory only when the text is dynamic.

No structure using $O(z)$ space offers good worst-case query times, because it is not known how to support fast access to the text within this space. The most recent theoretical bounds are due to Bille et al. (2017), who showed how we can store an $O(z \log(n/z) \log \log z)$-space data structure such that later we can locate all the occurrences of a pattern of length m in $O(m)$ time plus $O(\log \log z)$ time per occurrence. The implementation of Kreft and Navarro (2013) uses $O(z)$ space and offers the best compression with query times that are competitive in practice (despite their $O(m^2 z)$ worst-case bound).

Several authors (Schneeberger et al. 2009; Wandelt et al. 2013; Ferrada et al. 2014; Procházka and Holub 2014; Rahn et al. 2014) have independently proposed ideas essentially equivalent to finding the primary occurrences using an FM-index built on the substrings of length 2ℓ centered around phrase boundaries, where ℓ is a parameter. Gagie and Puglisi (2015) surveyed early work in this direction, and others (Danek et al. 2014; Wandelt and Leser 2015; Valenzuela 2016; Valenzuela and Mäkinen 2017; Ferrada et al. 2018) have given new implementations. These indexes are practical, though they have good worst-case query times only for patterns of length $m \leq \ell$. Increasing ℓ at construction worsens compression, whereas searching for patterns longer than ℓ requires the use of heuristics.

CDAWGs

A string $S[1..k]$ appearing s times in T tends to induce large isomorphic subtrees in the suffix tree of T, namely, the subtrees (with s leaves) of the suffix tree nodes representing the strings $S[1..k]$, $S[2..k]$, and so on. By regarding the suffix tree as a deterministic automaton, identifying all of its leaves with a single final state, and minimizing it, one obtains the CDAWG of T (Blumer et al. 1987). The minimization gets rid of the isomorphic subtrees, so the CDAWG benefits from repetitiveness as well.

The CDAWG can be represented using $O(e)$ space, where e is the number of right extensions of maximal repeats in T. With it, we can locate all the occurrences of a pattern of length m in $O(m \log \log n)$ time plus $O(1)$ time per occurrence (Belazzougui et al. 2015). Its disadvantage is that e is always larger than r and z, by a wide margin in practice, and hence CDAWGs tend to be much larger than the structures described above. The fastest variant (Belazzougui and Cunial 2017) uses $O(e + \overline{e})$ space (where $\overline{e}$ is the e measure of the reversed text) and answers locating queries in optimal time: $O(m)$ and then $O(1)$ per occurrence.

An advantage of CDAWGs is that they can be augmented to support suffix tree functionality (Belazzougui and Cunial 2017; Takagi et al. 2017), which is more powerful than just counting and locating pattern occurrences and of interest in bioinformatics applications. The CDAWG implements a number of suffix tree navigation operations in $O(\log n)$ time. Other compressed suffix trees for repetitive collections (Abeliuk et al. 2013; Navarro and Ordóñez 2016) build on a run-length encoded CSA or FM-index and apply grammar compression to the extra suffix tree components: the suffix tree's topology and the longest common prefix array. Like the CDAWG, they profit from repetitiveness but are significantly larger than their underlying compressed suffix array. The only compressed suffix tree of this kind offering good space guarantees (Gagie et al. 2017b), uses $O(r \log(n/r))$ space, and implements the suffix tree operations in time $O(\log(n/r))$.

Graph-Based Indexes

Graph-based indexes are the newest approach to compressed indexing of repetitive textual datasets so far focused on genomics (Eggertsson et al. 2017; Novak et al. 2017b). Most are generalizations of FM-indexes; Na et al.'s suffix tree of an alignment (Na et al. 2013a) and suffix array of an alignment (Na et al. 2013b) have now been superseded by their FM-index of an alignment (Na et al. 2016; Na et al. 2018). Work in this direction began with Ferragina et al. (2009) extending FM-indexes to labelled trees, followed by Bowe et al. (2012) extending them to de Bruijn graphs and Sirén et al. (2011, 2014) and Sirén (2017) extending them to labelled directed acyclic graphs (DAGs). Gagie

et al. (2017a) recently introduced a framework unifying these results.

Sirén et al.'s indexes are meant for pan-genomics and work essentially by embedding and indexing the DAGs in a kind of de Bruijn graph, allowing matches in recombinations of the input genomes. On the other hand, Na et al. index only the input genomes, essentially by embedding and indexing them in a kind of colored de Bruijn graph.

Both techniques depend on the presence of long, perfectly conserved regions to obtain better compression than run-length compressed FM-indexes, so it is not clear whether they will retain a significant advantage when compressing databases of tens of thousands of genomes in which low-frequency variations are included, how they compare to reference-free alignment methods, and to what extent they have applications outside bioinformatics.

Implementations

The run-length compressed FM-index of Mäkinen et al. (2010) is available at http://jltsiren.kapsi.fi/rlcsa. There are several implementations specifically for genomics (e.g., BGT, BWT-merge, MSBWT, RopeBWT2, SGA), many of which are available on GitHub. The index of Gagie et al. (2018) is implemented at https://github.com/nicolaprezza/r-index. The best implementation of the index of Kreft and Navarro (2013) is by Claude et al. (2016), currently available at https://github.com/migumar2/uiHRDC/tree/master/self-indexes/LZ.

Valenzuela (2016) and Valenzuela and Mäkinen (2017) give a good general-purpose hybrid index and a good pan-genomic aligner based on a hybrid index, available at https://www.cs.helsinki.fi/u/dvalenzu/software.

The DAG-based implementation of Sirén et al. is available at https://github.com/jltsiren/gcsa2. There are also several implementations of graph indexes specifically for genomics (Dilthey et al. 2015; Maciuca et al. 2016; Novak et al. 2017a; Paten et al. 2017), some of which are available at https://github.com/vgteam/vg.

Example Application

The main applications of indexing repetitive texts have been in bioinformatics, such as indexing genomic databases to support fast alignments. We may be given a database of a thousand human genomes, for example, and asked to preprocess it such that later, given a pattern and an integer k, we can quickly report which genomes contain substrings within edit distance k of that pattern. There are several techniques for implementing such approximate pattern matching using indexes for exact matching (Navarro and Raffinot 2002; Ohlebusch 2013; Mäkinen et al. 2015).

Future Directions for Research

There are several important open problems in the area. An immediate one is to explore the practical impact of the new run-length compressed FM-index of Gagie et al. (2018). In the longer term, combining run-length compressed FM-indexes with graph-based indexes and finding new applications for such indexes seem promising. Another challenge is to upgrade suffix arrays to suffix trees for repetitive collections; suffix trees require access to arbitrary suffix array cells, which is not known to be possible within $O(r)$ space, where r is again the number of runs in the BWT. Obtaining better search performance within space close to $O(z)$ in Lempel-Ziv- and grammar-based indexes is also important, where z is again the number of phrases in the LZ77 parse, because z is in practice significantly smaller than r, even if in theory they are incomparable. String attractors (Kempa and Prezza 2017) were introduced as a generalization of these two and many other measures of repetitiveness, so indexes based on them could be good with respect to all these measures simultaneously.

Cross-References

▸ Genomic Data Compression
▸ Grammar-Based Compression
▸ Inverted Index Compression

References

Abeliuk A, Cánovas R, Navarro G (2013) Practical compressed suffix trees. Algorithms 6(2):319–351

Belazzougui D, Cunial F (2017) Representing the suffix tree with the CDAWG. In: Proceedings of the 28th symposium on combinatorial pattern matching (CPM), pp 7:1–7:13

Belazzougui D, Cunial F, Gagie T, Prezza N, Raffinot M (2015) Composite repetition-aware data structures. In: Proceedings of the 26th symposium on combinatorial pattern matching (CPM), pp 26–39

Belazzougui D, Cunial F, Gagie T, Prezza N, Raffinot M (2017) Flexible indexing of repetitive collections. In: Proceedings of the 13th conference on computability in Europe (CiE), pp 162–174

Bille P, Ettienne MB, Gørtz IL, Vildhøj HW (2017) Time-space trade-offs for Lempel-Ziv compressed indexing. In: Proceedings of the 28th symposium on combinatorial pattern matching (CPM), pp 16:1–16:17

Blumer A, Blumer J, Haussler D, McConnell RM, Ehrenfeucht A (1987) Complete inverted files for efficient text retrieval and analysis. J ACM 34(3): 578–595

Bowe A, Onodera T, Sadakane K, Shibuya T (2012) Succinct de Bruijn graphs. In: Proceedings of the 12th workshop on algorithms in bioinformatics (WABI), pp 225–235

Claude F, Navarro G (2011) Self-indexed grammar-based compression. Fundamenta Informaticae 111(3): 313–337

Claude F, Navarro G (2012) Improved grammar-based compressed indexes. In: Proceedings of the 19th symposium on string processing and information retrieval (SPIRE), pp 180–192

Claude F, Fariña A, Martínez-Prieto MA, Navarro G (2016) Universal indexes for highly repetitive document collections. Inf Syst 61:1–23

Danek A, Deorowicz S, Grabowski S (2014) Indexes of large genome collections on a PC. PLoS One 9(10):e109384

Dilthey A, Cox C, Iqbal Z, Nelson MR, McVean G (2015) Improved genome inference in the MHC using a population reference graph. Nat Genet 47(6): 682–688

Do HH, Jansson J, Sadakane K, Sung W (2014) Fast relative Lempel-Ziv self-index for similar sequences. Theor Comput Sci 532:14–30

Eggertsson HP et al (2017) Graphtyper enables population-scale genotyping using pangenome graphs. Nat Genet 49(11):1654–1660

Ferrada H, Gagie T, Hirvola T, Puglisi SJ (2014) Hybrid indexes for repetitive datasets. Phil Trans R Soc A 372(2016):20130137

Ferrada H, Kempa D, Puglisi SJ (2018) Hybrid indexing revisited. In: Proceedings of the 20th workshop on algorithm engineering and experiments (ALENEX), pp 1–8

Ferragina P, Manzini G (2000) Opportunistic data structures with applications. In: Proceedings of the 41st symposium on foundations of computer science (FOCS), pp 390–398

Ferragina P, Manzini G (2005) Indexing compressed text. J ACM 52(4):552–581

Ferragina P, Luccio F, Manzini G, Muthukrishnan S (2009) Compressing and indexing labeled trees, with applications. J ACM 57(1):4:1–4:33

Gagie T, Puglisi SJ (2015) Searching and indexing genomic databases via kernelization. Front Bioeng Biotechnol 3:12

Gagie T, Gawrychowski P, Kärkkäinen J, Nekrich Y, Puglisi SJ (2012) A faster grammar-based self-index. In: Proceedings of the 6th conference on language and automata theory and applications (LATA), pp 240–251

Gagie T, Gawrychowski P, Kärkkäinen J, Nekrich Y, Puglisi SJ (2014) LZ77-based self-indexing with faster pattern matching. In: Proceedings of the 11th Latin American symposium on theoretical informatincs (LATIN), pp 731–742

Gagie T, Manzini G, Sirén J (2017a) Wheeler graphs: a framework for BWT-based data structures. Theor Comput Sci 698:67–78

Gagie T, Navarro G, Prezza N (2017b) Optimal-time text indexing in BWT-runs bounded space. Technical report 1705.10382, arXiv.org

Gagie T, Navarro G, Prezza N (2018) Optimal-time text indexing in BWT-runs bounded space. In: Proceedings of the 29th symposium on discrete algorithms (SODA), pp 1459–1477

Grossi R, Vitter JS (2000) Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In: Proceedings of the 32nd symposium on theory of computing (STOC), pp 397–406

Grossi R, Vitter JS (2005) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J Comput 35(2): 378–407

Kärkkäinen J, Ukkonen E (1996) Lempel-Ziv parsing and sublinear-size index structures for string matching. In: Proceedings of the 3rd South American workshop on string processing (WSP), pp 141–155

Kempa D, Prezza N (2017) At the roots of dictionary compression: string attractors. In: Proceedings of the 50th symposium on theory of computing (STOC), 2018. CoRR abs/1710.10964

Kreft S, Navarro G (2013) On compressing and indexing repetitive sequences. Theor Comput Sci 483: 115–133

Langmead B, Trapnell C, Pop M, Salzberg SL (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol 10(3):R25

Li H, Durbin R (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics 25(14):1754–1760

Maciuca S, del Ojo Elias C, McVean G, Iqbal Z (2016) A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In: Proceedings of the 16th workshop on algorithms in bioinformatics (WABI), pp 222–233

Mäkinen V, Navarro G, Sirén J, Välimäki N (2010) Storage and retrieval of highly repetitive sequence collections. J Comput Biol 17(3):281–308

Mäkinen V, Belazzougui D, Cunial F, Tomescu AI (2015) Genome-scale algorithm design: biological sequence analysis in the era of high-throughput sequencing. Cambridge University Press, Cambridge

Maruyama S, Nakahara M, Kishiue N, Sakamoto H (2013) ESP-index: a compressed index based on edit-sensitive parsing. J Discrete Algorithms 18:100–112

Na JC, Park H, Crochemore M, Holub J, Iliopoulos CS, Mouchard L, Park K (2013a) Suffix tree of alignment: an efficient index for similar data. In: Proceedings of the 24th international workshop on combinatorial algorithms (IWOCA), pp 337–348

Na JC, Park H, Lee S, Hong M, Lecroq T, Mouchard L, Park K (2013b) Suffix array of alignment: a practical index for similar data. In: Proceedings of the 20th symposium on string processing and information retrieval (SPIRE), pp 243–254

Na JC, Kim H, Park H, Lecroq T, Léonard M, Mouchard L, Park K (2016) FM-index of alignment: a compressed index for similar strings. Theor Comput Sci 638:159–170

Na JC, Kim H, Min S, Park H, Lecroq T, Léonard M, Mouchard L, Park K (2018) FM-index of alignment with gaps. Theor Comput Sci. https://doi.org/10.1016/j.tcs.2017.02.020

Navarro G (2017) A self-index on block trees. In: Proceedings of the 17th symposium on string processing and information retrieval (SPIRE), pp 278–289

Navarro G, Ordóñez A (2016) Faster compressed suffix trees for repetitive text collections. J Exp Algorithmics 21(1):article 1.8

Navarro G, Raffinot M (2002) Flexible pattern matching in strings – practical on-line search algorithms for texts and biological sequences. Cambridge University Press, Cambridge, UK

Nishimoto T, Tomohiro I, Inenaga S, Bannai H, Takeda M (2016) Dynamic index and LZ factorization in compressed space. In: Proceedings of the prague stringology conference (PSC), pp 158–170

Novak AM, Garrison E, Paten B (2017a) A graph extension of the positional Burrows-Wheeler transform and its applications. Algorithms Mol Biol 12(1):18:1–18:12

Novak AM et al (2017b) Genome graphs. Technical report 101378, bioRxiv

Ohlebusch E (2013) Bioinformatics algorithms: sequence analysis, genome rearrangements, and phylogenetic reconstruction. Oldenbusch Verlag, Bremen, Germany

Paten B, Novak AM, Eizenga JM, Garrison E (2017) Genome graphs and the evolution of genome inference. Genome Res 27(5):665–676

Procházka P, Holub J (2014) Compressing similar biological sequences using FM-index. In: Proceedings of the data compression conference (DCC), pp 312–321

Rahn R, Weese D, Reinert K (2014) Journaled string tree – a scalable data structure for analyzing thousands of similar genomes on your laptop. Bioinformatics 30(24):3499–3505

Sadakane K (2000) Compressed text databases with efficient query algorithms based on the compressed suffix array. In: Proceedings of the 11th international symposium on algorithms and computations (ISAAC), pp 410–421

Sadakane K (2003) New text indexing functionalities of the compressed suffix arrays. J Algorithms 48(2):294–313

Schneeberger K, Hagmann J, Ossowski S, Warthmann N, Gesing S, Kohlbacher O, Weigel D (2009) Simultaneous alignment of short reads against multiple genomes. Genome Biol 10(9):R98

Sirén J (2017) Indexing variation graphs. In: Proceedings of the 19th workshop on algorithm engineering and experiments (ALENEX), pp 13–27

Sirén J, Välimäki N, Mäkinen V (2011) Indexing finite language representation of population genotypes. In: Proceedings of the 11th workshop on algorithms in bioinformatics (WABI), pp 270–281

Sirén J, Välimäki N, Mäkinen V (2014) Indexing graphs for path queries with applications in genome research. IEEE/ACM Trans Comput Biol Bioinform 11(2):375–388

Takabatake Y, Tabei Y, Sakamoto H (2014) Improved ESP-index: a practical self-index for highly repetitive texts. In: Proceedings of the 13th symposium on experimental algorithms (SEA), pp 338–350

Takabatake Y, Nakashima K, Kuboyama T, Tabei Y, Sakamoto H (2016) siEDM: an efficient string index and search algorithm for edit distance with moves. Algorithms 9(2):26

Takagi T, Goto K, Fujishige Y, Inenaga S, Arimura H (2017) Linear-size CDAWG: new repetition-aware indexing and grammar compression. In: Proceedings of the 24th symposium on string processing and information retrieval (SPIRE), pp 304–316

Valenzuela D (2016) CHICO: a compressed hybrid index for repetitive collections. In: Proceedings of the 15th symposium on experimental algorithms (SEA), pp 326–338

Valenzuela D, Mäkinen V (2017) CHIC: a short read aligner for pan-genomic references. Technical report 178129, bioRxiv.org

Wandelt S, Leser U (2015) MRCSI: compressing and searching string collections with multiple references. Proc VLDB Endowment 8(5):461–472

Wandelt S, Starlinger J, Bux M, Leser U (2013) RCSI: scalable similarity search in thousand(s) of genomes. Proc VLDB Endowment 6(13):1534–1545

Ziv J, Lempel A (1977) A universal algorithm for sequential data compression. IEEE Trans Inf Theory 23(3):337–343

Compressed Representations for Complex Networks

▸ (Web/Social) Graph Compression

Computational Needs of Big Data

▸ Parallel Processing with Big Data

Computer Architecture for Big Data

Behrooz Parhami
Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA

Synonyms

Big data hardware acceleration; Hardware considerations for big data

Definitions

How features of general-purpose computer architecture impact big-data applications and, conversely, how requirements of big data lead to the emergence of new hardware and architectural support.

Overview

Computer architecture (Parhami 2005) is a subdiscipline of computer science and engineering that is concerned with designing computing structures to meet application requirements effectively, economically, reliably, and within prevailing technological constraints. In this entry, we discuss how features of general-purpose computer architecture impacts big-data applications and, conversely, how requirements of big data lead to the emergence of new hardware and architectural support.

Historical Trends in Computer Architecture

The von Neumann architecture for stored-program computers, with its single or unified memory, sometimes referred to as the Princeton architecture, emerged in 1945 (von Neumann 1945; von Neumann et al. 1947) and went virtually unchallenged for decades. It dominated the alternative Harvard architecture with separate program and data memories (Aiken and Hopper 1946) from the outset as the more efficient and versatile way of implementing digital computers.

As the workload for general-purpose computers began to change, adjustments in, and alternatives to, von Neumann architecture were proposed. Examples include de-emphasizing arithmetic operations in favor of data movement primitives, as seen in input/output and stream processors (Rixner 2001); introducing hardware aids for frequently used operations, as in graphic processing units or GPUs (Owens et al. 2008; Singer 2013); and adding special instructions for improved performance on multimedia workloads (Lee 1995; Yoon et al. 2001).

Recently, data-intensive applications necessitated another reassessment of the match between prevalent architectures and application requirements. The performance penalty of data having to be brought into the processor and sent back to memory through relatively narrow transfer channels was variously dubbed the "von Neumann bottleneck" (Markgraf 2007) and the "memory wall" (McKee 2004; Wulf and McKee 1995). Memory data transfer rates are measured in GB/s in modern machines, whereas the processing rates can be three or more decimal orders of magnitude higher.

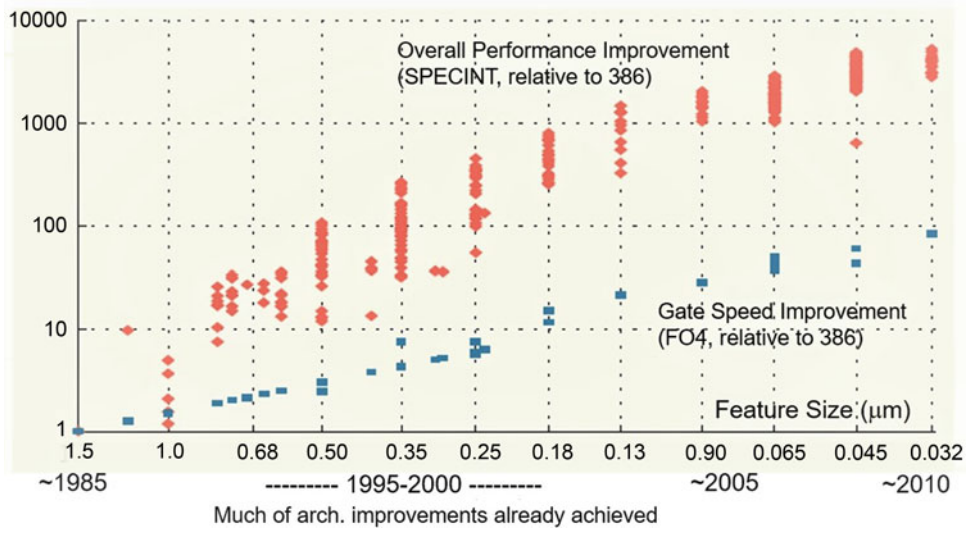
The term "non-von" (Shaw 1982) was coined to characterize a large category of machines that relaxed one or more of the defining features of the von Neumann architecture, so as to alleviate some of the perceived problems. Use of cache memories (Smith 1982), often in multiple levels, eased the von Neumann bottleneck for a while, but the bottleneck reemerged, as the higher cache data transfer bandwidth became inadequate and applications that lacked or had relatively limited locality of reference emerged. Memory interleaving and memory-access pipelining, pioneered by IBM (Smotherman 2010) and later used extensively in Cray supercomputers, was the next logical step.

Extrapolating a bit from Fig. 1 (which covers the period 1985–2010), and using round numbers, the total effect of architectural innovations has been a 100-fold gain in performance, on top of another factor-of-100 improvement due to faster gates and circuits (Danowitz et al. 2012). Both growth rates in Fig. 1 show signs of slowing down, so that future gains to support the rising processing need of big data will have to come, at least in part, from other sources. In the technology arena, use of emerging technologies will provide some boost for specific applications. An intriguing option is resurrecting hybrid digital/analog computing, which was sidelined long ago in favor of all-digital systems. Architecturally, specialization is one possible avenue for maintaining the performance growth rate, as are massively parallel and in-memory or near-memory computing.

How Big Data Affects Computer Architecture

The current age of big data (Chen and Zhang 2014; Hu et al. 2014) has once again exposed the von Neumann bottleneck, forcing computer architects to seek new solutions to the age-old problem, which has become much more serious. Processing speed continues to rise exponentially, while memory bandwidth increases at a much slower pace.

It is by now understood that big data is different from "lots of data." It is sometimes defined in terms of the attributes of volume, variety, velocity, and veracity, known as the "4Vs" (or "5 Vs," if we also include value). Dealing with big data requires big storage, big-data processing capability, and big communication bandwidth. The first two (storage and data processing) directly affect the architecture of the nodes holding



Computer Architecture for Big Data, Fig. 1 Technology advances and architectural innovations each contributed a factor of ∼100 improvement in processor performance over three decades. (Danowitz et al. 2012)

and processing the data. The part of communication that is internode is separately considered in this encyclopedia. However, there is also the issue of intranode communication represented in buses and networks-on-chip that belong to our architectural discussion here.

In addition to data volume, the type of data to be handled is also changing from structured data, as reflected, for example, in relational databases, to semi-structured and unstructured data. While this change has some negative effects in terms of making traditional and well-understood database technologies obsolete, it also opens up the possibility of using scalable processing platforms made of commodity hardware as part of the cloud-computing infrastructure. Massive unstructured data sets can be stored in distributed file systems, such as the ones designed in connection with Hadoop (Shafer et al. 2010) or SQL/noSQL (Cattell 2011).

In addition to the challenges associated with rising storage requirements and data access bandwidth, the processing load grows with data volume because of various needs. These are:

- Encryption and decryption
- Compression and decompression
- Sampling and summarization
- Visualization and graphing
- Sorting and searching
- Indexing and query processing
- Classification and data mining
- Deduction and learning

The first four items above, which are different forms of data translation, are discussed in the next section. The other items, viz., data transformations, will be discussed subsequently.

Architectural Aids to Data Translations

Many important data translations are handled by endowing a general-purpose architecture with suitable accelerator units deployed as coprocessors. Such accelerators are ideally custom integrated circuits, whose designs are fully optimized for their intended functions. However, in view of rapid advances in capabilities, performance, and energy efficiency of field-programmable gate arrays (Kuon et al. 2008), a vast majority of modern accelerators reported in the literature are built on FPGA circuits.

Accelerators for encryption and decryption algorithms have a long history (Bossuet et al. 2013). The binary choice of custom-designed VLSI or general-purpose processing for cryptographic computations has expanded to include a variety of intermediate solutions which include the use of FPGAs and GPUs. The best solution for an application domain depends not only on the required data rates and the crypto scheme, but also on power, area, and reliability requirements.

Data compression (Storer 1988) allows us to trade processing time and resources for savings in storage requirements. While any type of data can be compressed (e.g., text compression), massive sizes of video files make them a prime target for compression. With the emergence of video compression standards (Le Gall 1991), much effort has been expended to implement the standards on special-purpose hardware (Pirsch et al. 1995), offering orders of magnitude speed improvement over general-purpose programmed implementations.

Both sampling and summarization aim to reduce data volume while still allowing the operations of interest to be performed with reasonable precision. An alternative to post-collection reduction of data volume is to apply compression during data collection, an approach that in the case of sensor data collection is known as compressive sensing (Baraniuk 2007). Compressive sensing, when applicable, not only saves on processing time but also reduces transmission bandwidth and storage requirements. There are some mathematical underpinnings common to compressive sensing techniques, but at the implementation level, the methods and algorithms are by and large application-dependent.

Data visualization (Ward et al. 2010) refers to the production of graphical representation of data for better understanding of hidden structures and relationships. It may provide the only reasonable hope for understanding massive amounts of data,

although machine learning is a complementary and competing method of late. Several visualization accelerators were implemented in the late 1990s (e.g., Scott et al. 1998), but the modern trend is to use FPGA and cluster-based methods.

Architectural Aids to Data Transformations

Sorting is an extremely important primitive that is time-consuming for large data sets. It is used in a wide array of contexts, which includes facilitating subsequent searching operations. It can be accelerated in a variety of ways, from building more efficient data paths and memory access schemes, in order to make conventional sorting algorithms run faster, to the extreme of using hardware sorting networks (Mueller et al. 2012; Parhami 1999).

Indexing is one of the most important operations for large data sets, such as those maintained and processed by Google. Indexing and query processing have been targeted for acceleration within large-scale database implementations (Casper and Olukotun 2014; Govindaraju et al. 2004). Given the dominance of relational databases in numerous application contexts, a variety of acceleration methods have been proposed for operations on such databases (e.g., Bandi et al. 2004). Hardware components used in realizing such accelerators include both FPGAs and GPUs.

Hardware aids for classification are as diverse as classification algorithms and their underlying applications. A prime example in Internet routing is packet classification (Taylor 2005), which is needed when various kinds of packets, arriving at extremely high rates, must be separated for appropriate handling. Modern hardware aids for packet classification use custom arrays for pipelined network processing, content-addressable memories (Liu et al. 2010), GPUs (Owens et al. 2008), or tensor processing units (Sato et al. 2017). Data mining, the process of generating new information by examining large data sets, has also been targeted for acceleration (Sklyarov et al. 2015), and it can benefit from similar highly parallel processing approaches. Also falling under such acceleration schemes are aids and accelerators for processing large graphs (Lee et al. 2017).

The earliest form of deduction engines were theorem provers. An important application of automatic deduction and proof is in hardware verification (Cyrluk et al. 1995). In recent years, machine learning has emerged as an important tool for improving the performance of conventional systems and for developing novel methods of tackling conceptually difficult problems. Game-playing systems (Chen 2016) constitute important testbeds for evaluating various approaches to machine learning and their associated hardware acceleration mechanisms. This is a field that has just started its meteoric rise and bears watching for future applications.

Memory, Processing, and Interconnects

In both general-purpose and special-purpose systems interacting with big data, the three interconnected challenges of providing adequate memory capacity, supplying the requisite processing power, and enabling high-bandwidth data movements between the various data-handling nodes must be tackled (Hilbert and Lopez 2011).

The memory problem can be approached using a combination of established and novel technologies, including nonvolatile RAM, 3D stacking of memory cells, processing in memory, content-addressable memory, and a variety of novel (nanoelectronics or biologically inspired) technologies. We won't dwell on the memory architecture in this entry, because the memory challenge is addressed in other articles (see the Cross-References).

Many established methods exist for increasing the data-handling capability of a processing node. The architectural nomenclature includes superscalar and VLIW organizations, collectively known as instruction-level parallelism (Rau and Fisher 1993), hardware multithreading (Eggers et al. 1997), multicore parallelism (Gepner and Kowalik 2006), domain-specific hardware accelerators (examples cited earlier in this entry),

transactional memory (Herlihy and Moss 1993), and SIMD/vector architectural or instruction-set extensions (Lee 1995; Yoon et al. 2001). A complete discussion of all these methods is beyond the scope of this entry, but much pertinent information can be found elsewhere in this encyclopedia.

Intranode communication is achieved through high-bandwidth bus systems (Hall et al. 2000) and, increasingly, for multicore processors and systems-on-chip, by means of on-chip networks (Benini and De Micheli 2002). Interconnection bandwidth and latency rank high, along with memory bandwidth, among hardware capabilities needed for effective handling of big-data applications, which are increasingly implemented using parallel and distributed processing. Considerations in this domain are discussed in the entry "Parallel Processing for Big Data."

Future Directions

The field of computer architecture has advanced for several decades along the mainstream line of analyzing general applications and making hardware faster and more efficient in handling the common case while being less concerned with rare cases which have limited impact on performance. Big data both validates and challenges this assumption. It validates it in the sense that certain data-handling primitives arise in all contexts, regardless of the nature of the data or its volume. It challenges the assumption by virtue of the von-Neumann bottleneck or memory-wall notions discussed earlier. The age of big data will speed up the process of trickling down of architectural innovations from supercomputers, which have always led the way, into servers or even personal computers, which now benefit from parallel processing and, in some cases, massive parallelism.

Several studies have been performed about the direction of computer architecture in view of new application domains and technological developments in the twenty-first century (e.g., Ceze et al. 2016; Stanford 2012). Since much of the processing schemes for big data will be provided through the cloud, directions of cloud computing and associated hardware acceleration mechanisms become relevant to our discussion here (Caulfield et al. 2016). Advanced graphics processors (e.g., Nvidia 2016) will continue to play a key role in providing the needed computational capabilities for data-intensive applications requiring heavy numerical calculations. Application-specific accelerators for machine learning (Sato et al. 2017), and, more generally, various forms of specialization, constitute another important avenue of architectural advances for the big-data universe.

Cross-References

- Energy Implications of Big Data
- Parallel Processing with Big Data
- Storage Hierarchies for Big Data
- Storage Technologies for Big Data

References

Aiken HH, Hopper GM (1946) The automatic sequence controlled calculator – I. Electr Eng 65(8–9):384–391

Bandi N, Sun C, Agrawal D, El Abbadi A (2004) Hardware acceleration in commercial databases: a case study of spatial operations. In: Proceedings of the international conference on very large data bases, Toronto, pp 1021–1032

Baraniuk R (2007) Compressive sensing. IEEE Signal Process Mag 24(4):118–121

Benini L, De Micheli G (2002) Networks on chips: a new SoC paradigm. IEEE Comput 35(1):70–78

Bossuet L, Grand M, Gaspar L, Fischer V, Gogniat G (2013) Architectures of flexible symmetric key crypto engines – a survey: from hardware coprocessor to multi-crypto-processor system on chip. ACM Comput Surv 45(4):41

Casper J, Olukotun K (2014) Hardware acceleration of database operations. In: Proceedings of ACM/SIGDA international symposium on field-programmable gate arrays, Monterey, CA, pp 151–160

Cattell R (2011) Scalable SQL and NoSQL data stores. ACM SIGMOD Rec 39(4):12–27

Caulfield AM et al (2016) A cloud-scale acceleration architecture. In: Proceedings of 49th IEEE/ACM international symposium on microarchitecture, Orlando, FL, pp 1–13

Ceze L, Hill MD, Wenisch TE (2016) Arch2030: a vision of computer architecture research over the next 15 years, Computing Community Consortium. http://cra.

org/ccc/wp-content/uploads/sites/2/2016/12/15447-CCC-ARCH-2030-report-v3-1-1.pdf
Chen JX (2016) The evolution of computing: AlphaGo. Comput Sci Eng 18(4):4–7
Chen CLP, Zhang C-Y (2014) Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf Sci 275:314–347
Cyrluk D, Rajan S, Shankar N, Srivas MK (1995) Effective theorem proving for hardware verification. In: Theorem provers in circuit design. Springer, Berlin, pp 203–222
Danowitz A, Kelley K, Mao J, Stevenson JP, Horowitz M (2012) CPU DB: recording microprocessor history. Commun ACM 55(4):55–63
Eggers SJ, Emer JS, Levy HM, Lo JL, Stamm RL, Tullsen DM (1997) Simultaneous multithreading: a platform for next-generation processors. IEEE Micro 17(5): 12–19
Gepner P, Kowalik MF (2006) Multi-core processors: new way to achieve high system performance. In: Proceedings of IEEE international symposium on parallel computing in electrical engineering, Bialystok, pp 9–13
Govindaraju NK, Lloyd B, Wang W, Lin M, Manocha D (2004) Fast computation of database operations using graphics processors. In: Proceedings of the ACM SIGMOD international conference on management of data, Paris, pp 215–226
Hall SH, Hall GW, McCall JA (2000) High-speed digital system design: a handbook of interconnect theory and design practices. Wiley, New York
Herlihy M, Moss JEB (1993) Transactional memory: architectural support for lock-free data structures. In: Proceedings of the international symposium on computer architecture, San Diego, CA, pp 289–300
Hilbert M, Lopez P (2011) The world's technological capacity to store, communicate, and compute information. Science 332:60–65
Hu H, Wen Y, Chua T-S, Li X (2014) Toward scalable systems for big data analytics: a technology tutorial. IEEE Access 2:652–687
Kuon I, Tessier R, Rose J (2008) FPGA architecture: survey and challenges. Found Trends Electron Des Autom 2(2):135–253
Le Gall D (1991) MPEG: a video compression standard for multimedia applications. Commun ACM 34(4): 46–58
Lee RB (1995) Accelerating multimedia with enhanced microprocessors. IEEE Micro 15(2):22–32
Lee J, Kim H, Yoo S, Choi K, Hofstee HP, Nam GJ, Nutter MR, Jamsek D (2017) ExtraV: boosting graph processing near storage with a coherent accelerator. Proc VLDB Endowment 10(12):1706–1717
Liu AX, Meiners CR, Torng E (2010) TCAM razor: a systematic approach towards minimizing packet classifiers in TCAMs. IEEE/ACM Trans Networking 18(2): 490–500
Markgraf JD (2007) The von Neumann Bottleneck. Online source that is no longer accessible (will find a replacement for this reference during revisions)
McKee SA (2004) Reflections on the memory wall. In: Proceedings of the conference on computing frontiers, Ischia, pp 162–167
Mueller R, Teubner J, Alonso G (2012) Sorting networks on FPGAs. Int J Very Large Data Bases 21(1):1–23
Nvidia (2016) Nvidia Tesla P100: infinite compute power for the modern data center – technical overview. On-line document. http://images.nvidia.com/content/tesla/pdf/nvidia-teslap100-techoverview.pdf. Accessed 18 Feb 2018
Owens JD et al (2008) GPU computing. Proc IEEE 96(5):879–899
Parhami B (1999) Chapter 7: Sorting networks. In: Introduction to parallel processing: algorithms and architectures. Plenum Press, New York, pp 129–147
Parhami B (2005) Computer architecture: from microprocessors to supercomputers. Oxford University Press, New York
Pirsch P, Demassieux N, Gehrke W (1995) VLSI architectures for video compression – a survey. Proc IEEE 83(2):220–246
Rau BR, Fisher JA (1993) Instruction-level parallel processing: history, overview, and perspective. J Supercomput 7(1–2):9–50
Rixner S (2001) Stream processor architecture. Kluwer, Boston
Sato K, Young C, Patterson D (2017) An in-depth look at Google's first tensor processing unit, google cloud big data and machine learning blog, May 12. On-line document. http://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu. Accessed 18 Feb 2018
Scott ND, Olsen DM, Gannett EW (1998) An overview of the visualize FX graphic accelerator hardware. Hewlett Packard J 49:28–29
Shafer J, Rixner S, Cox AL (2010) The Hadoop distributed filesystem: balancing portability and performance. In: Proceedings of the IEEE international symposium on performance analysis of systems & software, White Plains, NY, pp 122–133
Shaw DE (1982) The non-von supercomputer, Columbia University technical report, on-line document. http://academiccommons.columbia.edu/catalog/ac:140914. Accessed 18 Feb 2018
Singer G (2013) The history of the modern graphics processor, TechSpot on-line article. http://www.techspot.com/article/650-history-of-the-gpu/. Accessed 18 Feb 2018
Sklyarov V et al (2015) Hardware accelerators for information retrieval and data mining. In: Proceedings of the IEEE conference on information and communication technology research, Bali, pp 202–205
Smith AJ (1982) Cache memories. ACM Comput Surv 14(8):473–530
Smotherman M (2010) IBM stretch (7030) – aggressive uniprocessor parallelism. On-line document. http://people.cs.clemson.edu/~mark/stretch.html. Accessed 18 Feb 2018
Stanford University (2012) 21st century computer architecture: a community white paper, on-line document.

http://csl.stanford.edu/~christos/publications/2012.21stcenturyarchitecture.whitepaper.pdf. Accessed 18 Feb 2018
Storer J (1988) Data compression. Computer Science Press, Rockville
Taylor DE (2005) Survey and taxonomy of packet classification techniques. ACM Comput Surv 37(3):238–275
von Neumann J (1945) First draft of a report on the EDVAC, University of Pennsylvania. On-line document. https://web.archive.org/web/20130314123032/http:/qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf. Accessed 14 Feb 2018
von Neumann J, Burks AW, Goldstine HH (1947) Preliminary discussion of the logical design of an electronic computing instrument. Institute for Advanced Study, Princeton
Ward MO, Grinstein G, Keim D (2010) Interactive data visualization: foundations, techniques, and applications. CRC Press, Natick
Wulf W, McKee S (1995) Hitting the wall: implications of the obvious. ACM Comput Archit News 23(1):20–24
Yoon C-W, Woo R, Kook J, Lee S-J, Lee K, Yoo H-J (2001) An 80/20-MHz 160-mW multimedia processor integrated with embedded DRAM, MPEG-4 accelerator, and 3-D rendering engine for mobile applications. IEEE J Solid State Circuits 36(11):1758–1767

Computer Security

▸ Big Data for Cybersecurity

Computing Average Distance

▸ Degrees of Separation and Diameter in Large Graphs

Computing the Cost of Compressed Data

Alistair Moffat and Matthias Petri
School of Computing and Information Systems, The University of Melboure, Melbourne, VIC, Australia

Synonyms

Data archiving; Data compression; Data retention

Definitions

Compression mechanisms reduce the storage cost of retained data. In the extreme case of data that must be retained indefinitely, the initial cost of performing the compression transformation can be amortized down to zero, since the savings in storage space continue to accrue without limit, albeit at decreasing rates as time goes by and disk storage becomes cheaper. A more typical scenario arises when a fixed data retention period must be supported, after which the stored data is no longer required; and when a certain level of access operations to the stored data can be expected, as part of a regulatory or compliance environment. In this second scenario, the *total cost of retention* (TCR) is a function of multiple competing factors, and the compression regime that provides the most compact storage might not be the one that provides the smallest TCR. This entry summarizes recent work in the area of cost models for data retention.

Overview

Data compression techniques have received extensive study over more than six decades. Entropy coding mechanisms such as Huffman, arithmetic, and asymmetric numeral system (ANS) coding have been used to support a range of modeling approaches, including those based on implicit and explicit dictionaries, those based on statistical prediction, those based on grammar identification, and those based on the Burrows-Wheeler transform. Witten et al. (1999) and Moffat and Turpin (2002) provide details of many such combinations; the ANS mechanism is more recent (Duda 2009, 2013; Moffat and Petri 2017). Each possible arrangement of model and coder represents another option for use by practitioners, with a wide range of trade-offs possible. Compression systems are typically compared in two quite different ways, based on the size of the compressed data (the *effectiveness* of the approach) and based on the computational and memory resources required to attain the encoding

and decoding transformation (the *efficiency* of the approach). Moreover, it is often the case that effectiveness and efficiency are in tension – that the most effective techniques are also the ones that are least efficient and vice versa. For example, software implementations of compression tools often include an optional parameter (in several cases, "–1" to "–9") to indicate the effort that should be used during encoding, with better compression resulting from greater effort; and principled trade-off mechanisms have also been proposed (Farruggia et al. 2014).

Hence, if a choice must be made between competing alternatives (compression programs or option settings) for use in some particular application or situation, a joint overall cost must be developed based on all relevant factors and used to measure the *total cost of retention*. This entry summarizes recent work by Liao et al. (2017) that describes such a cost model as a response to a *service-level agreement* and provides a synopsis of their key findings.

Data Storage and Retention

An archiving service stores different kinds of objects such as files, log entries, or variable-size data blobs. Adding objects to the storage system is generally referred to as a PUT request; and to retrieve objects, a GET request specifying one or more objects is issued to the storage system.

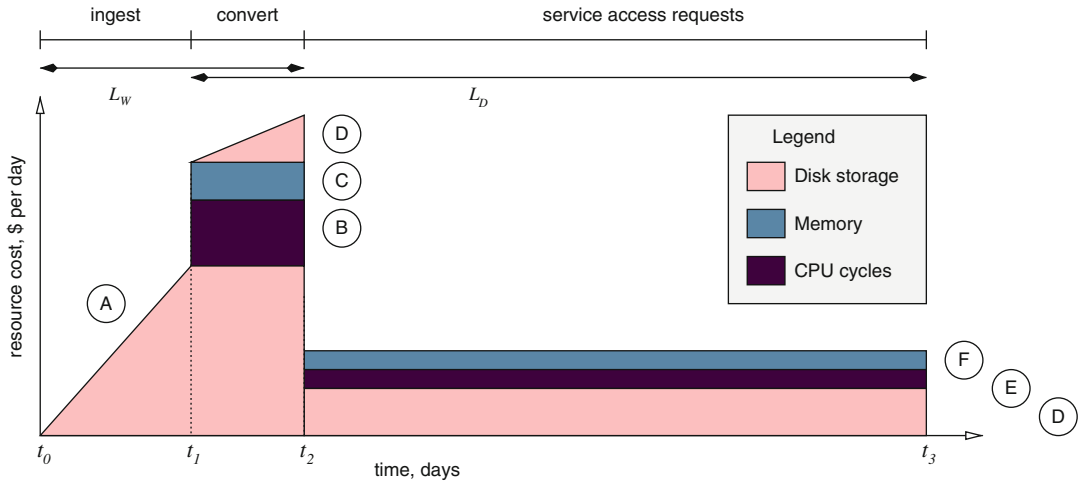
The objects stored by the archiving system can be viewed as a stream of bytes arriving at a constant rate when averaged over time. Liao et al. (2017) consider the case of data that arrives at an ingest rate of λ GiB/day, is subject to some obligatory minimum retention period of L_D days, and while retained must be available to GET requests assumed to occur at the rate of q queries per stored GiB per day. Each GET request for an object that was ingested during some particular day less than L_D days in the past is translated to a byte location within that day's data, and a request for the corresponding object (some number of bytes or kibibytes) is then issued, so that the object can be retrieved and delivered. For example, log data or transaction data might accumulate through some business process and be required to be retained for some minimum compliance period as part of a regulatory framework; and small fragments of that data might be required to be available on demand for audit or verification purposes. Such data retention regimes would typically be the subject of a *service-level agreement*, or SLA, including requirements in regard to the maximum delay L_W between ingest of any data and when it becomes available for querying operations, in regard to the maximum latency L_R permissible for any access operations, and in regard to other such performance and reliability requirements.

A cloud provider offering a data storage service naturally seeks to *minimize* the overall cost of providing the service within the constraints of the SLA. Compression is a key technology in this regard, since for long retention periods, the flagfall cost of compressing the data can almost certainly be recouped via decreased storage costs – provided, that is, the required GET operations can also be supported within the constraints imposed by the SLA.

Cost Model for Data Retention

Liao et al. (2017) propose that the total cost of a storage system be subdivided into four categories broadly aligned with the components measured by most cloud system providers: (a) permanent storage, (b) working memory, (c) compute cycles, and (d) I/O operations. Each of these categories is then measured and billed over a specific time period, with the units of each category converted into a common currency of dollars. For example, consuming 50 GiB of permanent storage for 12 months might be billed at $0.30 per GiB-month for a total storage cost of $180; and the compute cycles necessary to support GET operations on that data might cost a further $50 over the course of the year.

The service provider can be assumed to form data *bales* at daily or sub-daily intervals during the ingestion period, each bale assembled from multiple PUT operations. Once accumulated, the bale is then processed as a sequence of inde-



Computing the Cost of Compressed Data, Fig. 1 The bale-based data retention regime described by Liao et al. (2017) (Figure 2 of that paper, with copyright held by the original four authors). The six marked zones are: (A) cost of disk space for incoming data; (B) cost of CPU for index construction, if applicable; (C) cost of memory space for index construction, if applicable; (D) cost of disk space for retained data; (E) cost of CPU for decoding and access operations during retention period; (F) cost of memory space during retention period, if applicable

pendently retrievable *blocks*, with the bale as a whole following the life cycle shown in Fig. 1. A generic compression algorithm, such as gzip, or a domain specific algorithm, such as MPEG, might be applied to each block, to reduce the amount of permanent storage required while that block is held through the retention period. Ideally a compression scheme which minimizes the overall balance between storage cost and resource expenditure should be chosen, subject to the constraints set by the SLA parameters. For example, the maximum write delay L_W and the maximum read latency L_R set constraints on the encoding and decoding speed of the compression algorithm and/or the size of the blocks that can be allowed. Similarly, the choice of permanent storage has implications on the types of compression algorithms which can comply with the SLA and also on the cost involved. For example, high-latency permanent storage devices will require a compression algorithm with fast decompression speed, in order to handle GET request within L_R.

That is, each possible algorithm requires a certain amount of compute and memory resources (zones (B) and (C) in Fig. 1) to convert incoming bales. After the bale is converted, it resides on secondary storage, incurring a constant storage cost (D) for the remaining lifetime of the bale. To service GET requests arriving at an average rate of q queries per day per GiB of stored data, the system consumes additional compute and memory resources (E), (F) in order to decompress blocks. Additional IO costs may arise when transferring data to/from permanent storage and to/from the storage system itself.

Implications of the Cost Model

Liao et al. measure encoding and decoding speeds for compression mechanisms in four broad groups:

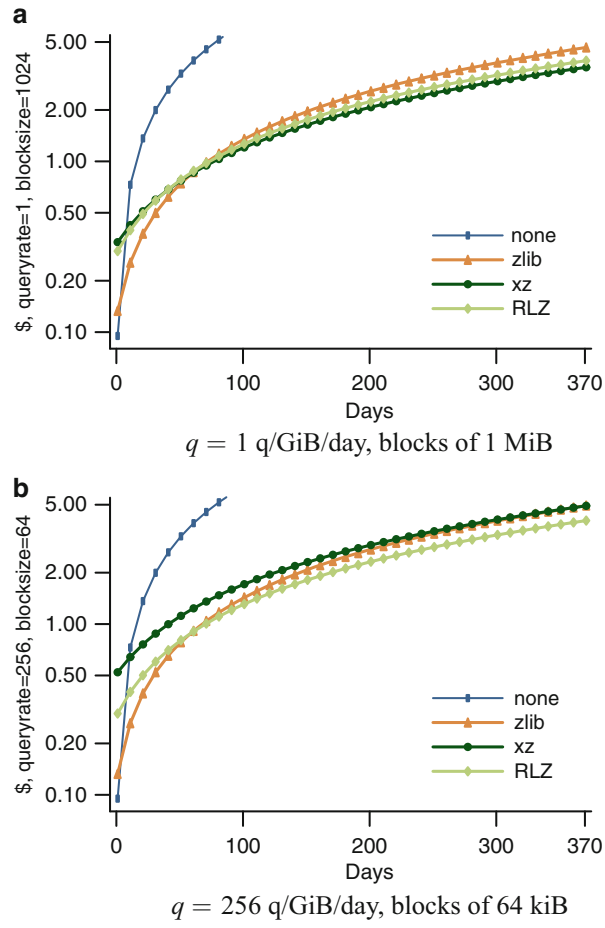
- No compression at all (denoted as method none);
- Fast encoding and decoding, based on an adaptive model and modest resource consumption, and with reasonably good compression effectiveness achieved at even relatively short block lengths (taking zlib, https://github.com/madler/zlib, as an exemplar);
- Excellent compression effectiveness when long blocks can be permitted, at the cost of more expensive encoding and decoding

(taking xz, http://tukaani.org/xz/, as an exemplar); and

- RLZ, a semi-static mechanism designed for fast decoding and good compression regardless of block size but requiring a per-bale memory-resident dictionary throughout each bale's retention period (Hoobin et al. 2011; Petri et al. 2015) (taking https://github.com/mpetri/rlz-store as an exemplar).

Liao et al. also provide compression effectiveness rates achieved for typical web data using a range of block sizes, highlighting the fact that for the adaptive zlib and xz approaches, compression effectiveness diminishes as blocks are made smaller.

Figure 2 takes that data and applies it to two different scenarios. In both graphs the total retention cost (TRC, in dollars per 64 GiB bale) is plotted on the vertical axis as a function of cumulative retention time, shown on the horizontal axis, for durations from 1 day to a little over 1 year. In the first pane, a low query rate $q = 1$ query per day per stored GiB is assumed. In this arrangement, relatively large blocks of 1 MiB can be employed as the encoding unit, and as a result xz achieves excellent compression rates, close to those of the semi-static RLZ mechanism. Moreover, the 8 MiB allocation of memory required by RLZ (to be precise, 8 MiB in the configuration plotted; dictionary size is another dimension that affects the comparison) is a cost drain that is not recouped even though RLZ decoding is faster. Hence, for all but short retention durations, where the encoding speed of zlib is an advantage, xz provides the smallest TRC.



Computing the Cost of Compressed Data, Fig. 2 Total retention cost for two different scenarios: (**a**) a low daily access rate allows large blocks to be used, favoring the "very good but somewhat slow" compression regime xz; (**b**) a high query rate requires small blocks, favoring the semi-static RLZ mechanism. For short retention periods, zlib is the cheapest option in both scenarios; and storing the data uncompressed (method none) is never an attractive option

In the lower pane, the query rate is considerably increased, to $q = 256$ GET requests per stored GiB per day, and the block sizes need to shorten (the value $b = 64$ kiB is used in the plot) so that decoding time doesn't swamp the other costs. Assuming that each GET request is to a different block, that query rate and block size mean that around 1.5% of the bale is decoded each day, and decoding throughput becomes a determining factor. Now RLZ's dictionary memory pays for itself, and it provides the best TRC, a combination of fast decoding and excellent compression effectiveness.

Based on the data they collected (including, as is also used in Figure 2, cost information from mid-2016 for a major cloud services provider), Liao et al. draw broad conclusions in regard to the efficacy of the various options:

- if the retention period is short, then zlib is the most economical choice, using a block size that decreases as the query rate increases;
- if the query rate is low, and if memory is expensive relative to secondary storage, then xz should be preferred; and
- if the query rate is high, and/or if the memory-to-disk cost ratio is more moderate, then RLZ-style approaches should be employed.

More generally, Liao et al. note that "thinking TRC" provides a framework in which compression improvements can be accurately measured and present one such approach that reduces RLZ-based retention costs by amortizing dictionary costs over a small number of consecutive bales.

Cross-References

- ▸ Energy Implications of Big Data
- ▸ Hardware-Assisted Compression
- ▸ Storage Hierarchies for Big Data

References

Duda J (2009) Asymmetric numeral systems. CoRR abs/0902.0271

Duda J (2013) Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. CoRR abs/1311.2540

Farruggia A, Ferragina P, Venturini R (2014) Bicriteria data compression: efficient and usable. In: Proceedings of the European symposium on algorithms (ESA), pp 406–417

Hoobin C, Puglisi SJ, Zobel J (2011) Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. PVLDB 5(3):265–273

Liao K, Moffat A, Petri M, Wirth A (2017) A cost model for long-term compressed data retention. In: Proceedings of the ACM international conference on web search and data mining (WSDM), pp 241–249

Moffat A, Petri M (2017) ANS-based index compression. In: Proceedings of the ACM international conference on information and knowledge management (CIKM), pp 677–686

Moffat A, Turpin A (2002) Compression and coding algorithms. Kluwer, Boston

Petri M, Moffat A, Nagesh PC, Wirth A (2015) Access time tradeoffs in archive compression. In: Proceedings of the Asia information retrieval societies conference (AIRS), pp 15–28

Witten IH, Moffat A, Bell TC (1999) Managing gigabytes: compressing and indexing documents and images. Morgan Kaufmann, San Francisco

Confidentiality

▸ Security and Privacy in Big Data Environment

Conflict-Free Replicated Data Types CRDTs

Nuno Preguiça[1], Carlos Baquero[2], and Marc Shapiro[3]

[1]NOVA LINCS and DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal

[2]HASLab/INESC TEC and Universidade do Minho, Braga, Portugal

[3]Sorbonne Université, LIP6 and INRIA, Paris, France

Definitions

A conflict-free replicated data type (CRDT) is an abstract data type, with a well-defined interface, designed to be replicated at multiple pro-

cesses and exhibiting the following properties: (i) any replica can be modified without coordinating with other replicas and (ii) when any two replicas have received the same set of updates, they reach the same state, deterministically, by adopting mathematically sound rules to guarantee state convergence.

Overview

Internet-scale distributed systems often replicate data at multiple geographic locations to provide low latency and high availability, despite outages and network failures. To this end, these systems must accept updates at any replica and propagate these updates asynchronously to the other replicas. This approach allows replicas to temporarily diverge and requires a mechanism for merging concurrent updates into a common state. CRDTs provide a principled approach to address this issue.

As any abstract data type, a CRDT implements some given functionality and exposes a well-defined interface. Applications interact with the CRDT only through this interface. As CRDTs are designed to be replicated and to allow uncoordinated updates, a key aspect of a CRDT is its semantics in the presence of concurrency. The concurrency semantics defines what is the behavior of the object in the presence of concurrent updates, defining the state of the object for any given set of received updates.

An application developer uses the CRDT interface and concurrency semantics to reason about the behavior of her application in the presence of concurrent updates. A system developer creating a system that provides CRDTs needs to focus on another aspect of CRDTs: the synchronization model. The synchronization model defines the requirements that the system must meet so that CRDTs work correctly. We now detail each of these aspects independently.

Concurrency Semantics

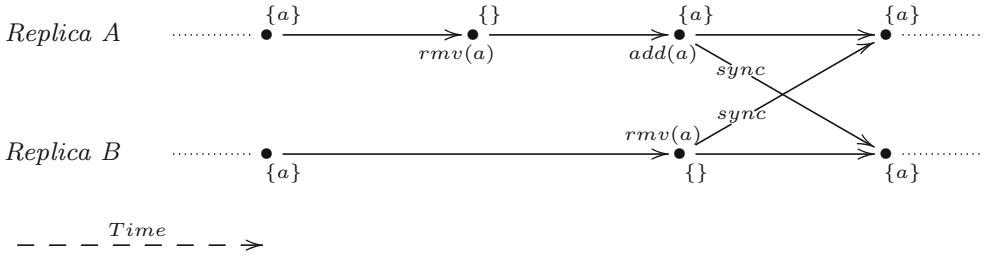
The operations defined in a data type may intrinsically commute or not. Consider, for instance, a counter data type, a shared integer that supports increment and decrement operations. As these operations commute (i.e., executing them in any order yields the same result), the counter data type naturally converges toward the expected result and reflects all executed operations reflects all executed operations.

Unfortunately, for most data types, this is not the case, and several concurrency semantics are reasonable, with different semantics being suitable for different applications. For instance, consider a shared memory cell supporting the assignment operation. If the initial value is 0, the correct outcome for concurrently assigning 1 and 2 is not well defined.

When defining the concurrency semantics, an important concept that is often used is that of the *happens-before* relation (Lamport 1978). In a distributed system, an event e_1 *happened-before* an event e_2, $e_1 \prec e_2$, iff (i) e_1 occurred before e_2 in the same process; or (ii) e_1 is the event of sending message m, and e_2 is the event of receiving that message; or (iii) there exists an event e such that $e_1 \prec e$ and $e \prec e_2$. When applied to CRDTs, we can say that an update u_1 *happened-before* an update u_2 iff the effects of u_1 had been applied in the replica where u_2 was initially submitted.

As an example, if an event was *Alice reserved the meeting room*, it is relevant to know if that was known when *Bob reserved the meeting room* to determine if Alice should be given priority or if the two users concurrently tried to reserve the same room.

For instance, let us use *happened-before* to define the semantics of the *add-wins* set (also known as observed-remove set, OR-set (Shapiro et al. 2011)). Intuitively, in the *add-wins* semantics, in the presence of two operations that do not commute, a concurrent add and remove of the same element, the add wins leading to a state where the element belongs to the set. More formally, the set interface has two update operations: (i) $\mathsf{add}(e)$, for adding element e to the set, and (ii) $\mathsf{rmv}(e)$, for removing element e from the set. Given a set of update operations O that are related by the happens-before partial order $\prec$, the state of the set is defined as $\{e \mid \mathsf{add}(e) \in O \wedge \nexists \mathsf{rmv}(e) \in O \cdot \mathsf{add}(e) \prec \mathsf{rmv}(e)\}$.



Conflict-Free Replicated Data Types CRDTs, Fig. 1 Run with an add-wins set

Figure 1 shows a run where an add-wins set is replicated in two replicas, with initial state $\{a\}$. In this example, in replica A, a is first removed and later added again to the set. In replica B, a is removed from the set. After receiving the updates from the other replica, both replicas end up with the element a in the set. The reason for this is that there is no $\mathsf{rmv}(a)$ that happened after the $\mathsf{add}(a)$ executed in replica A.

An alternative semantics based on the happens-before relation is *remove-wins*. Intuitively, in *remove-wins* semantics, in the presence of a concurrent add and remove of the same element, the remove wins leading to a state where the element is not in the set. More formally, given a set of update operations O, the state of the set is defined as: $\{e \mid \mathsf{add}(e) \in O \wedge \forall \mathsf{rmv}(e) \in O \cdot \mathsf{rmv}(e) \prec \mathsf{add}(e)\}$. In the previous example, after receiving the updates from the other replica, the state of both replicas would be the empty set, because there is no $\mathsf{add}(a)$ that happened after the $\mathsf{rmv}(a)$ in replica B.

Another relation that can be useful for defining the concurrency semantics is that of a total order among updates and particularly a total order that approximates wall-clock time. In distributed systems, it is common to maintain nodes with their physical clocks loosely synchronized. When combining the clock time with a site identifier, we have unique timestamps that are totally ordered. Due to the clock skew among multiple nodes, although these timestamps approximate an ideal global physical time, they do not necessarily respect the happens-before relation. This can be achieved by combining physical and logical clocks, as shown by Hybrid Logical Clocks (Kulkarni et al. 2014), or by only arbitrating a wall-clock total order for the events that are concurrent under causality (Zawirski et al. 2016).

This relation allows to define the *last-writer-wins* semantics, where the value written by the last writer wins over the values written previously, according to the defined total order. More formally, with the set O of operations now totally ordered by $\prec$, the state of a *last-writer-wins* set would be defined as: $\{e \mid \mathsf{add}(e) \in O \wedge \forall \mathsf{rmv}(e) \in O \cdot \mathsf{rmv}(e) \prec \mathsf{add}(e)\}$. Returning to our previous example, the state of the replicas after the synchronization would include a if, according the total order defined among the operations, the $\mathsf{rmv}(a)$ of replica B is smaller than the $\mathsf{add}(a)$ of replica A. Otherwise, the state would be the empty set.

We now briefly introduce the concurrency semantics proposed for several CRDTs.

Set

For a set CRDT, we have shown the difference between three possible concurrency semantics: *add-wins*, *remove-wins*, and *last-writer-wins*.

Register

A register CRDT maintains an opaque value and provides a single update operation that writes an arbitrary value: $\mathsf{wr}(value)$. Two concurrency semantics have been proposed leading to two different CRDTs: the *multi-value* register and the *last-writer-wins* register. In the *multi-value* register, all concurrently written values are kept. In this case, the read operation returns the set of concurrently written values. Formally, the state of a multi-value register is defined as the multi-set: $\{v \mid \mathsf{wr}(v) \in O \wedge \nexists \mathsf{wr}(u) \in O \cdot \mathsf{wr}(v) \prec \mathsf{wr}(u)\}$.

In the *last-writer-wins* register, only the value of the last write is kept, if any. Formally, the state of a last-writer-wins register can be defined as a set that is either empty or holds a single value: $\{v \mid \mathsf{wr}(v) \in O \wedge \nexists \mathsf{wr}(u) \in O \cdot \mathsf{wr}(v) \prec \mathsf{wr}(u)\}$, assuming some initial write.

Counter

A counter CRDT maintains an integer and can be modified by update operations inc and dec, to increase and decrease by one unit its value, respectively (this can easily generalize to arbitrary amounts). As mentioned previously, as operations intrinsically commute, the natural concurrency semantics is to have a final state that reflects the effects of all registered operations. Thus, the result state is obtained by counting the number of increments and subtracting the number of decrements: $|\{\mathsf{inc} \mid \mathsf{inc} \in O\}| - |\{\mathsf{dec} \mid \mathsf{dec} \in O\}|$.

Now consider that we want to add a write operation $\mathsf{wr}(n)$, to update the value of the counter to a given value. This opens two questions related with the concurrency semantics. First, what should be the final state when two concurrent write operations are executed? In this case, the last-writer-wins semantics would be simple (as maintaining multiple values, as in the multi-value register, is overly complex).

Second, what is the result when concurrent writes and inc/dec operations are executed? In this case, by building on the happens-before relation, we can define several concurrency semantics. One possibility is a *write-wins* semantics, where inc/dec operations have no effect when executed concurrently with the last write. Formally, for a given set O of updates that include at least a write operation, let v be the value in the last write, i.e., $\mathsf{wr}(v) \in O \wedge \nexists \mathsf{wr}(u) \in O \cdot \mathsf{wr}(v) \prec \mathsf{wr}(u)$. The value of the counter would be $v + o$, with $o = |\{\mathsf{inc} \mid \mathsf{inc} \in O \wedge \mathsf{wr}(v) \prec \mathsf{inc}\}| - |\{\mathsf{dec} \mid \mathsf{dec} \in O \wedge \mathsf{wr}(v) \prec \mathsf{dec}\}|$ representing inc/dec operations that happened after the last write.

Other CRDTs

A number of other CRDTs have been proposed in the literature, including CRDTs for elementary data structures, such as lists (Preguiça et al. 2009; Weiss et al. 2009; Roh et al. 2011), maps (Brown et al. 2014; Almeida et al. 2018), graphs (Shapiro et al. 2011), and more complex structures, such as JSON documents (Kleppmann and Beresford 2017). For each of these CRDTs, the developers have defined and implemented a type-specific concurrency semantics.

Synchronization Model

A replicated system needs to synchronize its replicas, by propagating and applying updates in every replica. There are two main approaches to propagate updates: state-based and operation-based replication.

In state-based replication, replicas synchronize by establishing bi-directional (or unidirectional) synchronization sessions, where both (one, resp.) replicas send their state to a peer replica. When a replica receives the state of a peer, it merges the received state with its local state. As long as the synchronization graph is connected, every update will eventually propagate to all replicas.

CRDTs designed for state-based replication define a merge function to integrate the state of a remote replica. It has been shown (Shapiro et al. 2011) that all replicas of a CRDT converge if (i) the states of the CRDT are partially ordered according to $\leq$ forming a join semilattice; (ii) an operation modifies the state s of a replica by an inflation, producing a new state that is larger or equal to the original state according to $\leq$, i.e., for any operation m, $s \leq m(s)$; and (iii) the merge function computes the join (least upper bound) of two states, i.e., for states s, u, it computes $s \sqcup u$.

In operation-based replication, replicas converge by propagating operations to every other replica. When an operation is received in a replica, it is applied to the local replica state. Besides requiring that every operation is reliably delivered to all replicas, e.g., by using some reliable multicast communication subsystem, some systems may require operations to be delivered according to some specific order, with causal order being the most common.

CRDTs designed for operation-based replication must define, for each operation, a generator

and an effector function. The generator function executes in the replica where the operation is submitted, the source replica, it has no side effects and generates an effector that encodes the side effects of the operation. In other words, the effector is a closure created by the generator depending on the state of the origin replica. The effector operation must be reliably executed in all replicas, where it updates the replica state. Shapiro et al. (2011) have shown that if effector operations are delivered in causal order, replicas will converge to the same state if concurrent effector operations commute. If effector operations may be delivered without respecting causal order, then all effector operations must commute. Most operation-based CRDT designs require causal delivery.

Alternative models: When operations modify only part of the state, propagating the complete state for synchronization to a remote replica is inefficient, as the remote replica already knows most of the state. Delta-state CRDTs (Almeida et al. 2018) address this issue by propagating only delta mutators, which encode the changes that have been made to a replica since the last communication. The first time a replica communicates with some other replica, the full state needs to be propagated. This can be improved by using a state summary (e.g., version vector) for computing and sending only the deltas in the first communication also, as shown in big delta-state CRDTs (van der Linde et al. 2016), typically at the cost of storing more metadata in the CRDT state. Another improvement is to compute digests that help determine which parts of a remote state are needed, avoiding shipping full states (Enes 2017).

In the context of operation-based replication, effector operations should be applied immediately to the source replica. However, propagation to other replicas can be deferred for some period and effectors stored in an outbound log, presenting an opportunity to compress the log by rewriting some operations – e.g., two add(1) operations in a counter can be converted in a add(2) operation. This mechanism has been used by Cabrita and Preguiça (2017). Delta mutators can also be seen as a compressed representation of a log of operations.

Operation-based CRDTs require executing a generator function against the replica state to compute an effector operation. In some scenarios, this may introduce an unacceptable delay for propagating an operation. Pure operation-based CRDTs (Baquero et al. 2014) address this issue by allowing the original operations to be propagated to all replicas, typically at the cost of more complex operations and of having to store more metadata in the CRDT state.

Key Research Findings

Preservation of Sequential Semantics

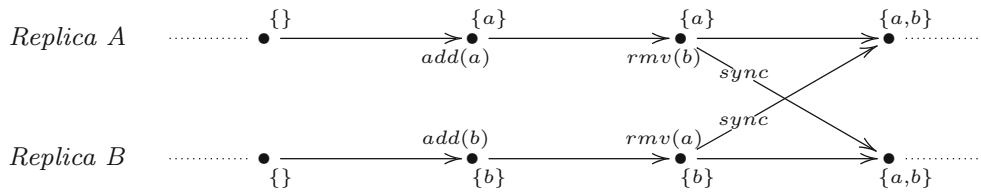
When modeling an abstract data type that has an established semantics under sequential execution, CRDTs should preserve that semantics. For instance, CRDT sets should ensure that if the last operation in a sequence of operations to a set added a given element, then a query operation immediately after that one will show the element to be present on the set. Conversely, if the last operation removed an element, then a subsequent query should not show its presence.

Sequential execution can occur even in distributed settings if synchronization is frequent. An instance can be updated in replica A, merged into another replica B and updated there, and merged back into replica A before A tries to update it again. In this case, we have a sequential execution, even though updates have been executed in different replicas.

Historically, not all CRDT designs have met this property. The *two-phase-set* CRDT (2PSet) does not allow re-adding an element that was removed, and thus it breaks the common sequential semantics. Later CRDT set designs, such as *add-wins* and *remove-wins* sets, do preserve the original sequential semantics while providing different concurrency semantics.

Extended Behavior Under Concurrency

Some CRDT designs handle concurrent operations by arbitrating a given sequential ordering to accommodate concurrent execution. For example, the state of a *last-writer-wins* set replica can be explained by a sequential execution of the op-



Conflict-Free Replicated Data Types CRDTs, Fig. 2 Add-wins set run showing that there might be no sequential execution of operations that explains CRDT behavior

erations according to the LWW total order used. When operations commute, such as in counters, there might even be several sequential executions that explain a given state.

Not all CRDTs need or can be explained by sequential executions. The add-wins set is an example of a CRDT where there might be no sequential execution of operations to explain the state observed, as Fig. 2 shows. In this example, the state of the set after all updates propagate to all replicas includes a and b, but in any sequential extension of the causal order, a remove operation would always be the last operation, and consequently the removed element could not belong to the set.

Some other CRDTs can exhibit states that are only reached when concurrency does occur. An example is the *multi-value register*. If used sequentially, sequential semantics is preserved, and a read will show the outcome of the most recent write in the sequence. However, if two or more values are written concurrently, the subsequent read will show all those values (as the *multi-value* name implies), and there is no sequential execution that can explain this result. We also note that a follow-up write can overwrite both a single value and multiple values.

Guaranties and Limitations

An important property of CRDTs is that an operation can always be accepted at any given replica and updates are propagated asynchronously to other replicas. In the CAP theorem framework (Brewer 2010; Gilbert and Lynch 2002), the CRDT conflict-free approach favors availability over consistency when facing communication disruptions. This leads to resilience to network failure and disconnection, since no prior coordination with other replicas is necessary before accepting an operation. Furthermore, operations can be accepted with minimal user perceived latency since they only require local durability. By eschewing global coordination, replicas evolve independently, and reads will not reflect operations accepted in remote replicas that have not yet been propagated to the local replica.

In the absence of global coordination, session guaranties (Terry et al. 1994) specify what the user applications can expect from their interaction with the system's interface. Both state-based CRDTs and operation-based CRDTs when supported by reliable causal delivery provide per-object causal consistency. Thus, in the context of a given replicated object, the traditional session guaranties are met. CRDT-based systems that lack transactional support can enforce system-wide causal consistency, by integrating multiple objects in a single map/directory object (Almeida et al. 2018). Another alternative is to use mergeable transactions to read from a causally consistent database snapshot and to provide write atomicity (Preguiça et al. 2014).

Some operations cannot be expressed in a conflict-free framework and will require global agreement. As an example, in an auction system, bids can be collected under causal consistency, and a new bid will only have to increase the offer with respect to bids that are known to causally precede it. However, closing the auction and selecting a single winning bid will require global agreement. It is possible to design a system that integrates operations with different coordination requirements and only resorts to global agreement when necessary (Li et al. 2012; Sovran et al. 2011).

Some global invariants, which usually are enforced with global coordination, can be enforced in a conflict-free manner by using escrow techniques (O'Neil 1986) that split the available resources among the different replicas. For instance, the Bounded Counter CRDT (Balegas et al. 2015b) defines a counter that never goes negative, among assigning to each replica a reserve of allowed decrements under the condition that the sum of all allowed decrements does not exceed the value of the counter. As long as its reserve is not exhausted, a replica can accept decrements without coordinating with other replicas. After a replica exhausts its reserve, a new decrement will either fail or require synchronizing with some replica that still can decrement. This technique uses point-to-point coordination and can be generalized to enforce other system-wide invariants (Balegas et al. 2015a)

Examples of Applications

CRDTs have been used in a large number of distributed systems and applications that adopt weak consistency models. The adoption of CRDTs simplifies the development of these systems and applications, as CRDTs guarantee that replicas converge to the same state when all updates are propagated to all replicas. We can group the systems and applications that use CRDTs into two groups: storage systems that provide CRDTs as their data model and applications that embed CRDTs to maintain their internal data.

CRDTs have been integrated in several storage systems that make them available to applications. An application uses these CRDTs to store its data, being the responsibility of the storage systems to synchronize the multiple replicas. The following commercial systems use CRDTs: Riak (Developing with Riak KV Data Types http://docs.basho.com/riak/kv/2.2.3/developing/data-types/.), Redis (Biyikoglu 2017), and Akka (Akka Distributed Data: https://doc.akka.io/docs/akka/2.5.4/scala/distributed-data.html.).
A number of research prototypes have also used CRDTs, including Walter (Sovran et al. 2011), SwiftCloud (Preguiça et al. 2014), and Antidote (Antidote: http://antidotedb.org.) (Akkoorath et al. 2016).

CRDTs have also been embedded in multiple applications. In this case, developers either used one of the available CRDT libraries, implemented themselves some previously proposed design, or designed new CRDTs to meet their specific requirements. An example of this latter use is Roshi (Roshi is a large-scale CRDT set implementation for timestamped events https://github.com/soundcloud/roshi.), a LWW-element-set CRDT used for maintaining an index in SoundCloud stream.

Future Directions of Research

Scalability

In order to track concurrency and causal predecessors, CRDT implementations often store metadata that grows linearly with the number of replicas (Charron-Bost 1991). While global agreement suffers from greater scalability limitations since replicas must coordinate to accept each operation, the metadata cost from causality tracking can limit the scalability of CRDTs when aiming for more than a few hundred replicas. A large metadata footprint can also impact the computation time of local operations and will certainly impact the required storage and communication.

Possible solutions can be sought in more compact causality representations when multiple replicas are synchronized among the same nodes (Malkhi and Terry 2007; Preguiça et al. 2014; Gonçalves et al. 2017) or by hierarchical approaches that restrict all to all synchronization and enable more compact mechanisms (Almeida and Baquero 2013).

Reversible Computation

Nontrivial Internet services require the composition of multiple subsystems, to provide storage, data dissemination, event notification, monitoring, and other needed components. When composing subsystems, which can fail independently or simply reject some operations, it is useful

to provide a CRDT interface that undoes previously accepted operations. Another scenario that would benefit from undo is collaborative editing of shared documents, where undo is typically a feature available to users.

Undoing an increment on a counter CRDT can be achieved by a decrement. Logoot-Undo (Weiss et al. 2010) proposes a solution for undoing (and redoing) operations for a sequence CRDT used for collaborative editing. However, providing a uniform approach to undoing, reversing, operations over the whole CRDT catalog is still an open research direction. The support of undo is also likely to limit the level of compression that can be applied to CRDT metadata.

Security

While access to a CRDT-based interface can be restricted by adding authentication, any accessing client has the potential to issue operations that can interfere with the other replicas. For instance, delete operations can remove all existing state. In state-based CRDTs, replicas have access to state that holds a compressed representation of past operations and metadata. By manipulation of this state and synchronizing to other replicas, it is possible to introduce significant attacks to the system operation and even its future evolution.

Applications that store state on third-party entities, such as in cloud storage providers, might not trust the provider and choose end-to-end encryption of the exchanged state. This, however, would require all processing to be done at the edge, under the application control. A research direction would be to allow some limited form of computation, such as merging state, over information whose content is subject to encryption. Potential techniques, such as homomorphic encryption, are likely to pose significant computational costs. An alternative is to execute operations on encrypted data without disclosing it, relying on specific hardware support, such as Intel SGX and ARM TrustZone.

Nonuniform Replicas

The replication of CRDTs typically assumes that eventually all replicas will reach the same state, storing exactly the same data. However, depending on the read operations available in the CRDT interface, it might not be necessary to maintain the same state in all replicas. For example, an object that has a single read operation returning the top-K elements added to the object only needs to maintain those top-K elements in every replica. The remaining elements are necessary if a remove operation is available, as one of the elements not in the top needs to be promoted when a top element is removed. Thus, each replica can maintain only the top-K elements and the elements added locally.

This replication model is named nonuniform replication (Cabrita and Preguiça 2017) and can be used to design CRDTs that exhibit important storage and bandwidth savings when compared with alternatives that keep all data in all replicas. Although it is clear that this model cannot be used for all data types, several useful CRDT designs have been proposed, including top-K, top-Sum, and histogram. To understand what data types can adopt this model and how to explore it in practice is an open research question.

Verification

An important aspect related with the development of distributed systems that use CRDTs is the verification of the correctness of the system. This involves not only verifying the correctness of CRDT designs but also the correctness of the system that uses CRDTs. A number of works have addressed these issues.

Regarding the verification of the correctness of CRDTs, several approaches have been taken. The most commonly used approach is to have proofs when designs are proposed or to use some verification tools for the specific data type, such as TLA (Lamport 1994) or Isabelle (Isabelle: http://isabelle.in.tum.de/.). There have also been some works that proposed general techniques for the verification of CRDTs (Burckhardt et al. 2014; Zeller et al. 2014; Gomes et al. 2017), which can be used by CRDT developers to verify the correctness of their designs. Some of these works (Zeller et al. 2014; Gomes et al. 2017) include specific frameworks that help the developer in the verification process.

A number of other works have proposed techniques to verify the correctness of distributed systems that use CRDTs (Gotsman et al. 2016; Zeller 2017; Balegas et al. 2015a). These works typically require the developer to specify the properties that the distributed system must maintain and a specification of the operations in the system (that is independent of the actual code of the system). Despite these works, the verification of the correctness of CRDT designs and of systems that use CRDTs, how these verification techniques can be made available to programmers, and how to verify the correctness of implementations remain an open research problem.

Acknowledgements This work was partially supported by NOVA LINCS (UID/CEC/04516/2013), EU H2020 LightKone project (732505), and SMILES line in project TEC4Growth (NORTE-01-0145-FEDER-000020).

References

Akkoorath DD, Tomsic AZ, Bravo M, Li Z, Crain T, Bieniusa A, Preguiça N, Shapiro M (2016) Cure: strong semantics meets high availability and low latency. In: Proceedings of the 2016 IEEE 36th international conference on distributed computing systems (ICDCS), pp 405–414. https://doi.org/10.1109/ICDCS.2016.98

Almeida PS, Baquero C (2013) Scalable eventually consistent counters over unreliable networks. CoRR abs/1307.3207. http://arxiv.org/abs/1307.3207, 1307.3207

Almeida PS, Shoker A, Baquero C (2018) Delta state replicated data types. J Parallel Distrib Comput 111:162–173. https://doi.org/10.1016/j.jpdc.2017.08.003

Balegas V, Duarte S, Ferreira C, Rodrigues R, Preguiça NM, Najafzadeh M, Shapiro M (2015a) Putting consistency back into eventual consistency. In: Réveillère L, Harris T, Herlihy M (eds) Proceedings of the tenth European conference on computer systems, EuroSys 2015, Bordeaux. ACM, pp 6:1–6:16. https://doi.org/10.1145/2741948.2741972

Balegas V, Serra D, Duarte S, Ferreira C, Shapiro M, Rodrigues R, Preguiça NM (2015b) Extending eventually consistent cloud databases for enforcing numeric invariants. In: 34th IEEE symposium on reliable distributed systems, SRDS 2015, Montreal. IEEE Computer Society, pp 31–36. https://doi.org/10.1109/SRDS.2015.32

Baquero C, Almeida PS, Shoker A (2014) Making operation-based CRDTs operation-based. In: Proceedings of the first workshop on principles and practice of eventual consistency, PaPEC'14. ACM, New York, pp 7:1–7:2. https://doi.org/10.1145/2596631.2596632

Biyikoglu C (2017) Under the hood: Redis CRDTs (conflict-free replicated data types). Online https://goo.gl/tGqU7h. Accessed 24 Nov 2017

Brewer E (2010) On a certain freedom: exploring the CAP space, invited talk at PODC 2010, Zurich

Brown R, Cribbs S, Meiklejohn C, Elliott S (2014) Riak DT map: a Composable, convergent replicated dictionary. In: Proceedings of the first workshop on principles and practice of eventual consistency, PaPEC'14. ACM, New York, pp 1:1–1:1. https://doi.org/10.1145/2596631.2596633

Burckhardt S, Gotsman A, Yang H, Zawirski M (2014) Replicated data types: specification, verification, optimality. In: Proceedings of the 41st ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'14. ACM, New York, pp 271–284. https://doi.org/10.1145/2535838.2535848

Cabrita G, Preguiça N (2017) Non-uniform replication. In: Proceedings of the 21th international conference on principles of distributed systems, OPODIS 2017, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, LIPIcs

Charron-Bost B (1991) Concerning the size of logical clocks in distributed systems. Inf Process Lett 39(1):11–16. https://doi.org/10.1016/0020-0190(91)90055-M

Enes V (2017) Efficient Synchronization of State-based CRDTs. Master's thesis, Universidade do Minho. http://vitorenesduarte.github.io/page/other/msc-thesis.pdf

Gilbert S, Lynch N (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2):51–59. https://doi.org/10.1145/564585.564601

Gomes VBF, Kleppmann M, Mulligan DP, Beresford AR (2017) Verifying strong eventual consistency in distributed systems. Proc ACM Program Lang 1(OOPSLA):109:1–109:28. https://doi.org/10.1145/3133933

Gonçalves RJT, Almeida PS, Baquero C, Fonte V (2017) DottedDB: anti-entropy without Merkle trees, deletes without tombstones. In: Proceedings of the 2017 IEEE 36th symposium on reliable distributed systems (SRDS), pp 194–203. https://doi.org/10.1109/SRDS.2017.28

Gotsman A, Yang H, Ferreira C, Najafzadeh M, Shapiro M (2016) 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'16. ACM, New York, pp 371–384. https://doi.org/10.1145/2837614.2837625

Kleppmann M, Beresford AR (2017) A conflict-free replicated JSON datatype. IEEE Trans Parallel Distrib Syst 28(10):2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

Kulkarni SS, Demirbas M, Madappa D, Avva B, Leone M (2014) Logical physical clocks. In: Aguilera MK, Querzoni L, Shapiro M (eds) Principles of distributed systems – 18th international conference, OPODIS

C

2014, Cortina d'Ampezzo. Proceedings. Lecture notes in computer science, vol 8878. Springer, pp 17–32. https://doi.org/10.1007/978-3-319-14472-6_2
Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565. https://doi.org/10.1145/359545.359563
Lamport L (1994) The temporal logic of actions. ACM Trans Program Lang Syst 16(3):872–923. https://doi.org/10.1145/177492.177726
Li C, Porto D, Clement A, Gehrke J, Preguiça N, Rodrigues R (2012) Making geo-replicated systems fast as possible, consistent when necessary. In: Proceedings of the 10th USENIX conference on operating systems design and implementation, OSDI'12. USENIX Association, Berkeley, pp 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906
Malkhi D, Terry DB (2007) Concise version vectors in winfs. Distrib Comput 20(3):209–219. https://doi.org/10.1007/s00446-007-0044-y
O'Neil PE (1986) The escrow transactional method. ACM Trans Database Syst 11(4):405–430. https://doi.org/10.1145/7239.7265
Preguiça N, Marques JM, Shapiro M, Letia M (2009) A commutative replicated data type for cooperative editing. In: Proceedings of the 2009 29th IEEE international conference on distributed computing systems, ICDCS'09. IEEE Computer Society, Washington, DC, pp 395–403. https://doi.org/10.1109/ICDCS.2009.20
Preguiça NM, Zawirski M, Bieniusa A, Duarte S, Balegas V, Baquero C, Shapiro M (2014) Swiftcloud: fault-tolerant geo-replication integrated all the way to the client machine. In: 33rd IEEE international symposium on reliable distributed systems workshops, SRDS workshops 2014, Nara. IEEE Computer Society, pp 30–33. https://doi.org/10.1109/SRDSW.2014.33
Roh HG, Jeon M, Kim JS, Lee J (2011) Replicated abstract data types: building blocks for collaborative applications. J Parallel Distrib Comput 71(3):354–368. https://doi.org/10.1016/j.jpdc.2010.12.006
Shapiro M, Preguiça N, Baquero C, Zawirski M (2011) Conflict-free replicated data types. In: Proceedings of the 13th international conference on stabilization, safety, and security of distributed systems, SSS'11. Springer, Berlin/Heidelberg, pp 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642
Sovran Y, Power R, Aguilera MK, Li J (2011) Transactional storage for geo-replicated systems. In: Proceedings of the twenty-third ACM symposium on operating systems principles, SOSP'11. ACM, New York, pp 385–400. https://doi.org/10.1145/2043556.2043592
Terry DB, Demers AJ, Petersen K, Spreitzer M, Theimer M, Welch BB (1994) Session guarantees for weakly consistent replicated data. In: Proceedings of the third international conference on parallel and distributed information systems, PDIS'94, Austin. IEEE Computer Society, pp 140–149. https://doi.org/10.1109/PDIS.1994.331722
van der Linde A, Leitão JA, Preguiça N (2016) Δ-CRDTs: making δ-CRDTs delta-based. In: Proceedings of the 2nd workshop on the principles and practice of consistency for distributed data, PaPoC'16. ACM, New York, pp 12:1–12:4. https://doi.org/10.1145/2911151.2911163
Weiss S, Urso P, Molli P (2009) Logoot: a scalable optimistic replication algorithm for collaborative editing on p2p networks. In: Proceedings of the 2009 29th IEEE international conference on distributed computing systems, ICDCS'09. IEEE Computer Society, Washington, DC, pp 404–412. https://doi.org/10.1109/ICDCS.2009.75
Weiss S, Urso P, Molli P (2010) Logoot-undo: distributed collaborative editing system on p2p networks. IEEE Trans Parallel Distrib Syst 21(8):1162–1174. https://doi.org/10.1109/TPDS.2009.173
Zawirski M, Baquero C, Bieniusa A, Preguiça N, Shapiro M (2016) Eventually consistent register revisited. In: Proceedings of the 2nd workshop on the principles and practice of consistency for distributed data, PaPoC'16. ACM, New York, pp 9:1–9:3. https://doi.org/10.1145/2911151.2911157
Zeller P (2017) Testing properties of weakly consistent programs with repliss. In: Proceedings of the 3rd international workshop on principles and practice of consistency for distributed data, PaPoC'17. ACM, New York, pp 3:1–3:5. https://doi.org/10.1145/3064889.3064893, https://dl.acm.org/authorize?N37605
Zeller P, Bieniusa A, Poetzsch-Heffter A (2014) Formal specification and verification of CRDTs. In: Formal techniques for distributed objects, FORTE 2014. Lecture notes in computer science. Springer, pp 33–48

Conformance Checking

Jorge Munoz-Gama
Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile, Santiago, Chile

Synonyms

Business process conformance checking

Definitions

Given an event log and a process model from the same process, conformance checking compares the recorded event data with the model to

identify commonalities and discrepancies. The conformance between a log and model can be quantified with respect to different quality dimensions: *fitness*, *precision*, and *generalization*.

Overview

Conformance checking compares an event log with a process model of the same process (Munoz-Gama 2016). An event log is composed of a series of log traces where each log trace relates to the sequence of observed events of a process instance, i.e., a case. An event can be related to a particular activity in the process but can also record many other process information such as time stamp, resource, and cost. In a real-life context, event logs can be extracted from Process-Aware Information Systems (PAIS) such as workflow management (WFM) systems, business process management (BPM) systems, or typical relational databases, such as SAP database. Similarly, process models can often be extracted from the organization's information systems. These can be normative models that the organization uses to manage their process, or descriptive, created by hand or automatically discovered to gain insight into their processes (van der Aalst 2013).

Depending on the nature of the model, discrepancies between the log and model can have different interpretations (van der Aalst 2016). For a normative model, deviations indicate violations of imposed constraints. For example, a banking process may require the processing and approval of a loan to be done by different employees to avoid the risk of misconduct (four-eyes principle). Clearly, conformance checking between an event log of the handled loan applications and the process model can be applied to assess compliance. On the other hand, for a descriptive model, deviations indicate that the model is not fully capturing all the observed behavior in the log. For example, process analysts might perform conformance checking on the models discovered by different process discovery algorithms before selecting the ones that are of sufficient quality for further analysis.

To illustrate conformance checking, a simple process is introduced. Figure 1 shows a doctoral scholarship application process in an informal modeling notation. This process consists of eight activities: *Start Processing*, *Evaluate Project*, *Evaluate Academic Record*, *Evaluate Advisor CV*, *Final Evaluation*, *Accept*, *Reject*, and *Notify Results*. To begin the process, an applicant has to submit their academic record, their advisor's CV, and a description of their proposed project. Once the required documents are received, the committee would begin by evaluating the submitted documents. As shown by the AND gateway, the committee can choose to evaluate the three documents in any order. Following the preliminary evaluation, a final evaluation is done to consolidate the previous results. This leads to either the acceptance or rejection of the application. Finally, the applicant is notified of the result. An example of log trace corresponding to an accepted application could be ⟨*Start Processing*, *Evaluate Project*, *Evaluate Academic Record*, *Evaluate Advisor CV*, *Final Evaluation*, *Accept*, *Notify Results*⟩.

Dimensions of Conformance

Through conformance checking, commonalities and discrepancies between a log and model are quantified. One simple idea would be to consider that a log and model are conforming with each other if the observed behavior in the log is captured by the model. This means that a log and model are perfectly conforming if all the log traces can be *fitted* to the model. However, this can be easily achieved with a model that allows any behavior. Such models do not provide much information to the data analyst about the process. This shows that there is a need to consider conformance with respect to different dimensions.

Currently, conformance is generally considered with respect to three dimensions – *fitness*, *precision*, and *generalization*.

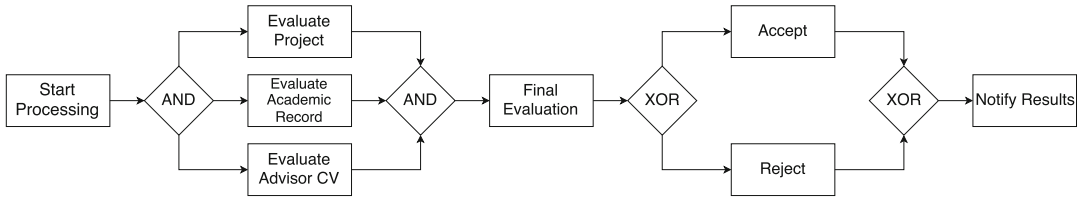
Fitness relates to how well a model and log fit each other. A log trace perfectly *fits* the model if it can be replayed onto the model and corresponds to a complete model trace. For example, ⟨*Start Processing*, *Evaluate Project*, *Evaluate Academic*

Record, *Evaluate Advisor CV*, *Final Evaluation*, *Accept*, *Notify Results*⟩ perfectly fits the model in Fig. 1 since each of the observed steps can be sequentially replayed at the model, and the trace corresponds to a particular possible way to execute the process model. However, the trace ⟨*Start Processing*, *Evaluate Project*, *Evaluate Academic Record*, *Final Evaluation*, *Reject*, *Notify Results*⟩ does not fit the model because the advisor's CV (*Evaluate Advisor CV*) is never evaluated. This suggests that the corresponding application has been rejected without proper evaluation.

Precision relates to a model's ability to capture the observed behavior without allowing unseen behavior. It is not enough to have a model that is perfectly fitting with the log since this can be easily achieved with a model that permits any behavior. Consider the "flower" model in Fig. 2; it consists of all the transitions attached to a state that corresponds to both the start and end state. This means any sequence involving the connected transitions is permissible by the model. Though perfectly fitting with the log, such underfitting model does not convey much useful information to the user. In contrary, the process model illustrated in Fig. 3 is much more precise than the flower model.

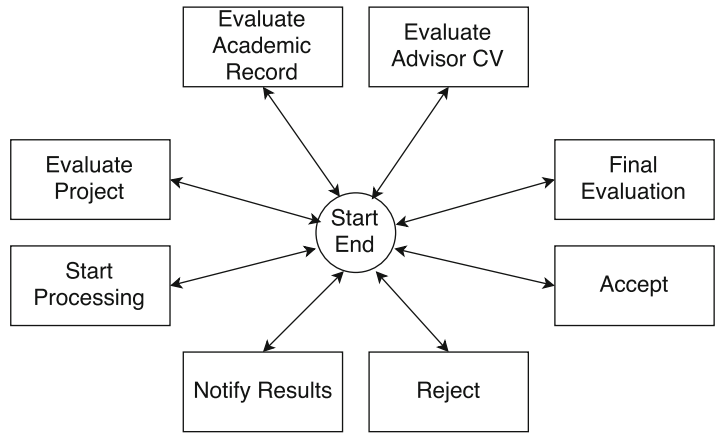
Generalization relates to a model's ability to account for yet to be observed behavior. Typically, an event log only represents a small fraction of the possible behavior in the process. As such, a good model must be generalizing enough so that unobserved but possible behavior is described. For example, if the model in Fig. 4 was discovered from an event log that contains only the trace ⟨*Start Processing*, *Evaluate Project*, *Evaluate Academic Record*, *Evaluate Advisor CV*, *Final Evaluation*, *Accept*, *Notify Results*⟩, then the model would be both perfectly fitting and precise since all observed behaviors are captured by the model and that it does not allow any unseen behavior. Clearly, there is much unseen behavior that is very likely to occur in the future, e.g., the rejection of an application. This shows that, while it is important to have precise models, it is also important to avoid overfitting the observed behavior.

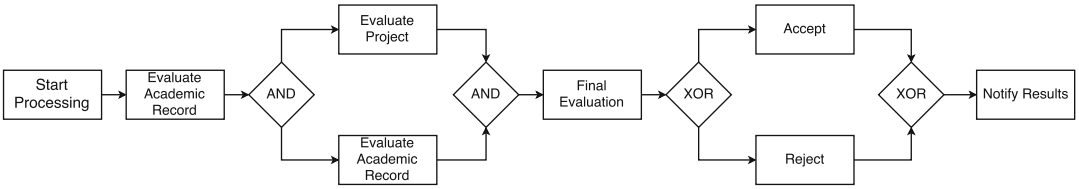
Some authors consider a fourth dimension called *simplicity*, relating to the model complexity, i.e., simple models should be preferred over



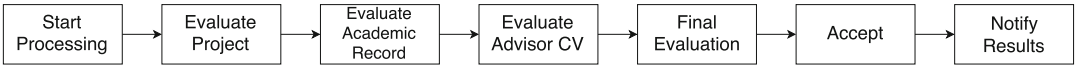
Conformance Checking, Fig. 1 Informal process model of a university scholarship process

Conformance Checking, Fig. 2 Imprecise flower model of the doctoral scholarship process

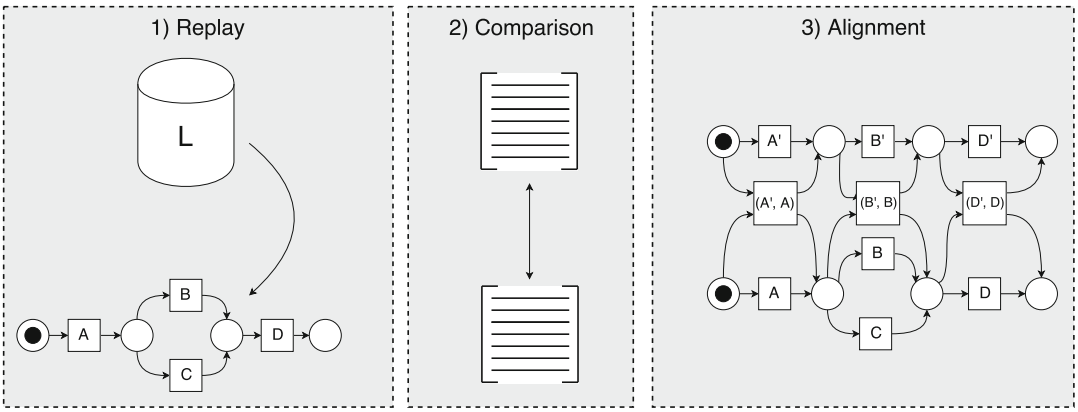




Conformance Checking, Fig. 3 Precise but unfitting model of the doctoral scholarship process



Conformance Checking, Fig. 4 Model that overfits one particular possible execution of the doctoral scholarship process



Conformance Checking, Fig. 5 Three main types of conformance checking approaches

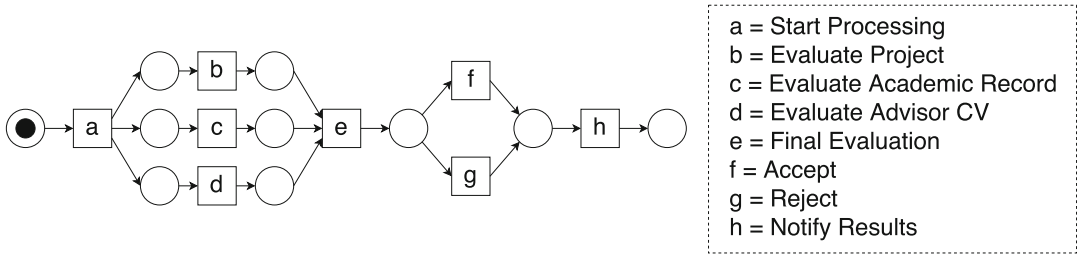
complex models if both describe the same behavior. However this quality dimension relates only to the model and therefore is not normally measured by conformance checking techniques. This dimension is covered in the *Automated Process Discovery* entry of this encyclopedia.

Overall, while the three quality dimensions are orthogonal to each other, in a real-life context, one is unlikely to find a pair of log and model that are in perfect conformance (i.e., perfectly fitting, precise, and generalizing). Often times, different scenarios may require different conformance levels and prioritization of the quality dimensions. For example, to analyze the well-established execution paths of a process, an analyst might prioritize fitness over the other dimensions. On the other hand, if an event log only contains a small number of cases, generalization would likely to be prioritized over the other dimensions to account for possible future behavior.

Types of Conformance

Conformance checking techniques can be applied to understand and quantify these relationships between a log and model. There is a large collection of approaches and metrics that are based on different ways to compare a log and model.

Figure 5 shows that there are three main groups of conformance checking approaches – *replay*, *comparison*, and *alignment*. Replay-based approaches replay log traces onto the model and record information about the conformance during the replay. Process models can be denoted in different modeling notations, e.g., Business Process Modeling Notation (BPMN), Petri nets, and process trees, and each



Conformance Checking, Fig. 6 Model M_1 of the university doctoral scholarship process denoted in Petri net notation

representation bias has distinct characteristics, e.g., formalism and determinism. However, a proper process model is typically executable so that log traces can be re-executed stepwise by the model. Comparison-based approaches convert both the log and model into a common representation so that the log and model are directly comparable. Last but not least, alignment-based approaches seek to explain observed behavior in terms of modeled behavior by aligning log traces with the model. This brings conformance checking to the level of events and can offer detailed diagnosis on conformance issues.

Key Research Findings

In this section, two conformance checking approaches and three conformance metrics are presented.

Token Replay

Token replay is a replay-based conformance checking approach that measures the fitness between a log and model by replaying log traces onto process models denoted in the Petri net notation (Rozinat and van der Aalst 2008).

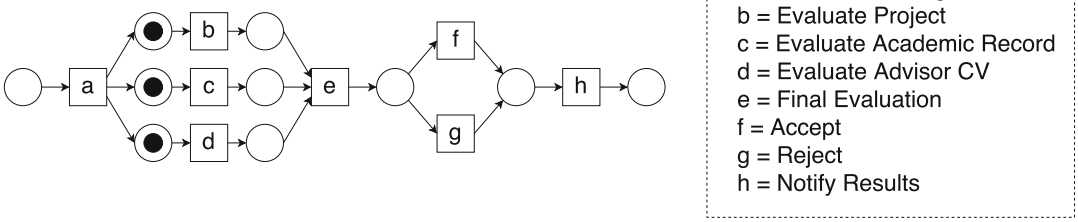
Consider model M_1 in Fig. 6 and log L_1 in Fig. 7. Model M_1 is denoted in Petri net notation so that the squares correspond to the activities in the process, filled circles correspond to tokens that mark the state of a process instance as activities get executed, and empty circles correspond to places that hold tokens. To execute an activity, all its input places (i.e., all the places connected by an incoming arrow to the activity) must have at least one token. This means that the activity

$$L_1 = [t_1 = \langle a, b, c, d, e, f, h\rangle,$$
$$t_2 = \langle a, c, b, e, f, h\rangle,$$
$$t_3 = \langle a, b, d, c, g, e, h\rangle]$$

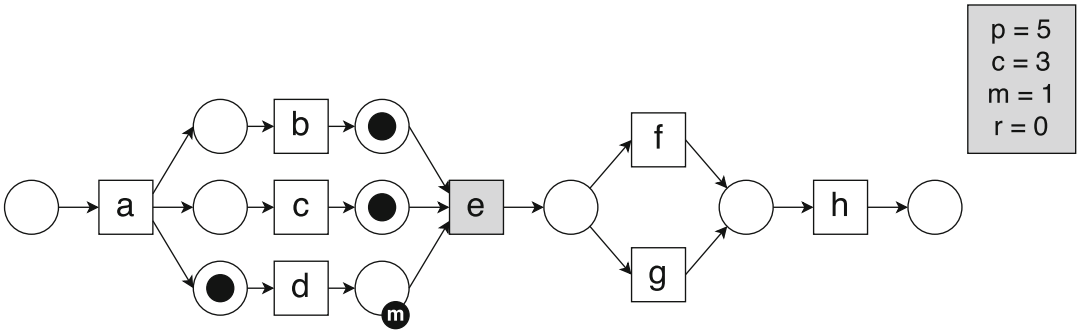
Conformance Checking, Fig. 7 Running example: event log L_1

is *enabled* and can be *fired*. When an activity is fired, the activity *consumes* a token from each of its input places before *producing* one token at each of its output places (i.e., all the places connected by an outgoing arrow from the activity). For example, activity a in model M_1 in Fig. 6 is currently enabled. If the activity is fired, it would consume the token of its input place and produce three tokens at each of its three output places as illustrated in Fig. 8. As such, an instance of the process can be recorded by successively firing enabled activities until no activities are enabled. For a valid Petri net model, an instance is initiated by having a token at each of the source places (i.e., places without any incoming arrows) and is deemed to be completed by firing activities until there is only one token at each of the sink places (i.e., places without any outgoing arrows) and none at any other places. The sequence of fired activities corresponds to a complete model trace, i.e., a possible execution of the model.

Log traces can be replayed onto the model by successively firing the activities related to each event in the log trace at an initiated Petri net model. If the log trace is perfectly fitting with the model, there should not be any problem with the replay since the log trace corresponds to a complete model trace. However, for deviating traces, replay would not be successful due to missing or



Conformance Checking, Fig. 8 Model M_1 after firing activity a



Conformance Checking, Fig. 9 Missing token to fire activity e in token replay of trace $t_2 = \langle a, c, b, e, f, h \rangle$

redundant tokens. An activity might be marked to be fired in the log trace but is not enabled in the model due to missing tokens at its input places. Consider the replay of trace t_2 in L_1. Starting from the initial state of model M_1 as shown in Fig. 6, the firing of activity a, b, and c would consume three tokens and produce five tokens in the process. Figure 9 shows the state of model M_1 after the firing of the first three activities and records the number of consumed and produced tokens. According to trace t_2, the next activity to be fired is activity e. However, this is not possible since one of the input places of activity e does not have any token, i.e., activity e is not enabled. To continue the replay, the missing token is artificially added into the empty input place, and the number of missing token is incremented. The rest of trace t_2 (activity f and g) can be replayed successively. Figure 10 shows that, after firing activity h, there is a remaining token in the input place of activity d since this activity was not fired in the replay of trace t_2. As recalled, a process instance is only completed when there is only one token at each of the sink places and none at any other places. To complete the replay, the remaining token is removed artificially, and the number of remaining tokens is incremented.

Based on the count of each token types consumed, produced, missing, and remaining (p = 8, c = 8, m = 1, r = 1), the fitness between model M_1 and trace t_2 can be computed as:

$$\begin{aligned} fitness(t_2, M) &= \frac{1}{2}(1 - \frac{m}{c}) + \frac{1}{2}(1 - \frac{r}{p}) \\ &= \frac{1}{2}(1 - \frac{1}{5}) + \frac{1}{2}(1 - \frac{1}{5}) = 0.8 \end{aligned}$$

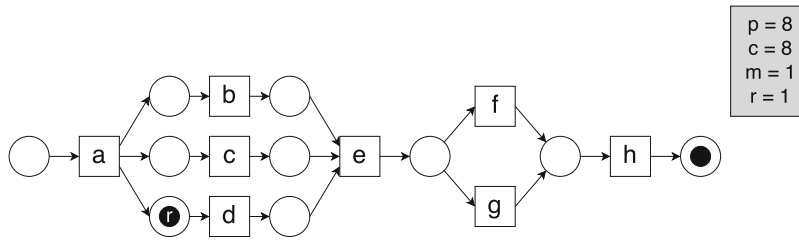
This fitness metric can be extended to the log level by considering the number of produced, consumed, missing, and remaining tokens from the token replay of all log traces.

Cost-Based Alignment

The token replay approach can easily identify deviating traces in an event log. Moreover, the deviation severity can be compared using a fitness metric computed from the number of produced, consumed, missing, and remaining tokens. However, the token replay approach is prone to creat-

ing too many tokens for highly deviating traces so that any behavior is allowed. This can lead to an overestimation of the fitness. The approach is also specific to the Petri net notation. More importantly, in the case of a deviating trace, the approach does not provide a model explanation of the log trace. For example, the deviations in trace $t_2 = \langle a, c, b, e, f, h \rangle$ can be explained if it was considered with respect to the complete model trace $\langle a, c, b, d, e, f, h \rangle$. From this mapping, it is clear that the log trace is missing the execution of activity d (Evaluate Advisor CV). These mappings from log traces to model traces were introduced as *alignments* to address this limitation (van der Aalst et al. 2012).

Alignments are tables of two rows where the top row corresponds to the observed behavior (i.e., log projection) and the bottom row corresponds to the modeled behavior (i.e., model projection). Each column is therefore a move in the alignment where the observed behavior is aligned with the modeled behavior. Consider alignment γ_1 in Fig. 11. This alignment aligns trace $t_3 = \langle a, b, d, c, g, e, h \rangle$ in L_1 and model M_1 in Fig. 6. The top row (ignoring $\gg$) yields the trace $t_3 = \langle a, b, d, c, g, e, h \rangle$, and the bottom row (ignoring $\gg$) yields a complete model trace $\langle a, b, d, c, e, g, h \rangle$. For each move in alignment γ_1, the top row matches the bottom row if the step in the log trace matches the step in the model trace. This is called a *synchronous move*. In the case of deviations, a no-move symbol $\gg$ is placed in the bottom row if there is a step in the log trace that cannot be mimicked by the model trace. For example, activity g is executed before activity e in trace t_3, but model M_1 requires activity e to be fired before activity g. Hence, a *log move* is put where the top row has activity g and the bottom row has a no-move $\gg$. Similarly, a no-move symbol $\gg$ is placed in the top row if there is a step in the model trace that cannot be mimicked by the log trace. For example, activity g is executed after activity e in the model trace according to model M_1. Therefore, a *model move* is added where the top row has a no-move $\gg$ and the bottom row has activity g. It is also possible



Conformance Checking, Fig. 10 Remaining token in token replay of trace $t_2 = \langle a, c, b, e, f, h \rangle$

$\gamma_1 =$

a	b	d	c	g	e	$\gg$	h
a	b	d	c	$\gg$	e	g	h

$\gamma_2 =$

a	b	d	c	$\gg$	g	e	$\gg$	h
a	b	$\gg$	c	d	$\gg$	e	g	h

$\gamma_3 =$

a	b	d	c	$\gg$	g	e	h
a	b	d	c	e	g	$\gg$	h

Conformance Checking, Fig. 11 Possible alignments between trace $t_3 = \langle a, b, d, c, g, e, h \rangle$ in L_1 and model M_1 in Fig. 6

$\gamma_4 =$

a	b	d	c	g	e	h	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$
$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	a	b	d	c	e	g	h

Conformance Checking, Fig. 12 Default alignment between trace t_3 in log L_1 and model M_1 in Fig. 6

that there are invisible transitions in the model which are not observable in the log. Similar to a model move, there would be a no-move in the top row and an invisible transition label in the bottom row. In total, there are four types of *legal moves* in an alignment: synchronous move, log move, model move, and invisible move.

For a particular log trace and model, there could be many possible alignments where each represents a different explanation of the observed behavior in terms of modeled behavior. For example, Fig. 11 shows three possible alignments between trace t_3 and model M_1 in Fig. 6. Clearly, alignment γ_1 and γ_3 are better alignments of trace t_3 and model M_1 than alignment γ_2 since they provide closer explanations with less log moves and model moves. The quality of an alignment can be quantified by assigning costs to moves. In general, model moves and log moves are assigned higher costs than synchronous moves because they represent deviations between modeled behavior and observed behavior. A standard cost assignment could be that all model moves and log moves are assigned a cost of 1 and synchronous moves and invisible moves are assigned a cost of 0. Invisible moves are normally assigned zero costs as they are related to invisible routing transitions in the model that are not observable in the log. Under the standard cost assignment, the costs of the alignments in Fig. 11 can be computed as follows:

$$cost(\gamma_1) = 0+0+0+0+1+0+1+0 = 2$$
$$cost(\gamma_2) = 0+0+1+0+1+1+0+1+0 = 4$$
$$cost(\gamma_3) = 0+0+0+0+1+0+1+0 = 2$$

This confirms the previous intuition that alignment γ_1 and γ_3 are better alignments than alignment γ_2. Alignments with the minimal costs correspond to *optimal alignments* thatgive the closest explanations of log traces in terms of modeled behavior. Note that there could be multiple optimal alignments for a particular log trace. For example, alignment γ_1 and γ_3 are both optimal alignments of trace t_3 under the standard cost assignment. Furthermore, optimal alignments are only optimal with respect to the given cost assignment. For example, alignment γ_1 would cease to be the optimal alignment if model moves and log moves of activity g are assigned a cost of 2 (i.e., $cost(\gamma_1) = 4$) to reflect that having deviations at the decision part of the process is quite severe. In practice, optimal alignments can be automatically found by finding the cheapest complete model trace in the synchronous product of the log trace and model using heuristic algorithms with proven optimality guarantees, e.g., the A* algorithm (van der Aalst et al. 2012).

Alignments can also be used to compute conformance metrics with respect to the different quality dimensions.

Cost-Based Fitness Metric

The fitness of a log trace and a model can be quantified by comparing the cost of an optimal alignment with the worst case scenario cost (Adriansyah 2014). In the worst scenario, the log trace is completely unfitting with the model. A default alignment between the two can be computed by assigning all the steps in the log trace as log moves and all the steps in the complete model trace as model moves. Since the optimal alignment minimizes the total alignment cost, the least costly complete model trace is used. Figure 12 shows the default alignment between trace t_3 and model M_1 under the standard cost assignment. The top row (ignoring $\gg$) yields trace t_3, and the bottom row (ignoring $\gg$) yields a complete model trace $\langle a, b, d, c, e, g, h\rangle$. The cost-based fitness of trace t_3 can be computed as:

$$\begin{aligned} fitness(t_3, M) &= 1 - \frac{cost(align(t_3, M))}{cost(align_{\text{default}}(t_3, M))} \\ &= 1 - \frac{0+0+0+0+1+0+1+0}{1+1+1+1+1+1+1+1+1+1+1+1+1+1} \\ &= 1 - \frac{2}{14} = 0.857 \end{aligned}$$

where a fitness value of 1.0 means that the model and log trace are perfectly fitting.

Escaping Arc Precision

Precision based on escaping arcs can also be computed using alignments (Adriansyah et al. 2012). As previously mentioned, an imprecise model allows unobserved behavior, i.e., underfitting. For example, consider the Petri net model M_1 in Fig. 6 and the optimal alignments (under the standard cost assignment) between model M_1 and log L_1 in Fig. 13. Clearly, model M_1 is not perfectly precise as it allows for behavior that is not observed in log L_1. According to model M_1, activity b, c, and d can be executed in parallel following the execution of activity a. However, none of the log traces execute activity d after activity a. This imprecision in the model can be quantified by constructing a prefix automaton using the model projection of the alignments, i.e., the bottom row of the alignments. As previously presented, model projections of alignments explain potentially unfitting log traces in terms of modeled behavior so that they can be replayed on the process model. Figure 14 illustrates the constructed prefix automaton $\mathcal{A}_1$ for the alignments between log L_1 and model M_1 (ignoring the circles highlighted in red for now). Each prefix of the model projections of the alignments identifies a state (represented as circles), and the numberin the states corresponds to the weight. For example, the state $\langle a \rangle$ has a weight of 3 because it appears three times in the model projections (all three alignments start with activity a). On the other hand, the state $\langle a, c, b \rangle$ is only present in alignment γ_6 and therefore has a weight of 1. The states of automaton $\mathcal{A}_1$ represent states reached by the model during the execution of the log. For any particular state in automaton $\mathcal{A}_1$, there might be activities that are enabled by the model but are not observed in the log execution. These activities indicate imprecisions of the model and are called *escaping arcs* of the model. Escaping arc states (represented as circles highlighted in red) are added to automaton $\mathcal{A}_1$ by replaying the automaton onto the model and checking for enabled activities at each state. For example, at state $\langle a \rangle$ (i.e., after firing activity a), activity b, c, and d are enabled as shown in Fig. 8. However, the prefix $\langle a, d \rangle$ was not observed in the construction of automaton $\mathcal{A}_1$ using log L_1. This means that there is an escaping arc from state $\langle a \rangle$ to state $\langle a, d \rangle$, and this is added to the automaton by the state highlighted in red. The rest of the escaping arcs can be added in a similar way.

With the constructed prefix automaton, escaping arc precision can be computed by comparing the number of escaping arcs with the number of allowed arcs for all states:

$$
\begin{aligned}
precision(\mathcal{A}_1) &= 1 - \frac{\sum_{s \in S} \omega(s) \cdot |esc(s)|}{\sum_{s \in S} \omega(s) \cdot |mod(s)|} \\
&= 1 - \frac{3 \cdot 0 + 3 \cdot 1 + 2 \cdot 0 + \ldots + 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 0}{3 \cdot 1 + 3 \cdot 3 + 2 \cdot 2 + \ldots + 1 \cdot 2 + 1 \cdot 1 + 1 \cdot 1} \\
&= 1 - \frac{6}{36} = 0.833
\end{aligned}
$$

where S is the set of states in automaton $\mathcal{A}_1$, $\omega(\cdot)$ maps a state $s \in S$ to its weight, $esc(\cdot)$ maps a state $s \in S$ to its set of escaping arc states, and $mod(\cdot)$ maps a state $s \in S$ to its set of allowed states. A precision value of 1.0 indicates perfect precision, i.e., the model only allows observed behavior and nothing else.

Artificial Negative Events

Another approach to measure precision is through artificial negative events. Artificial negative events are induced by observing events that did not occur in the event log. These unobserved events (i.e., negative events) give information about things that are not allowed to

Conformance Checking, Fig. 13 Optimal alignments between trace t_1, t_2, t_3 in log L_1 and model M_1 in Fig. 6

γ_5 =

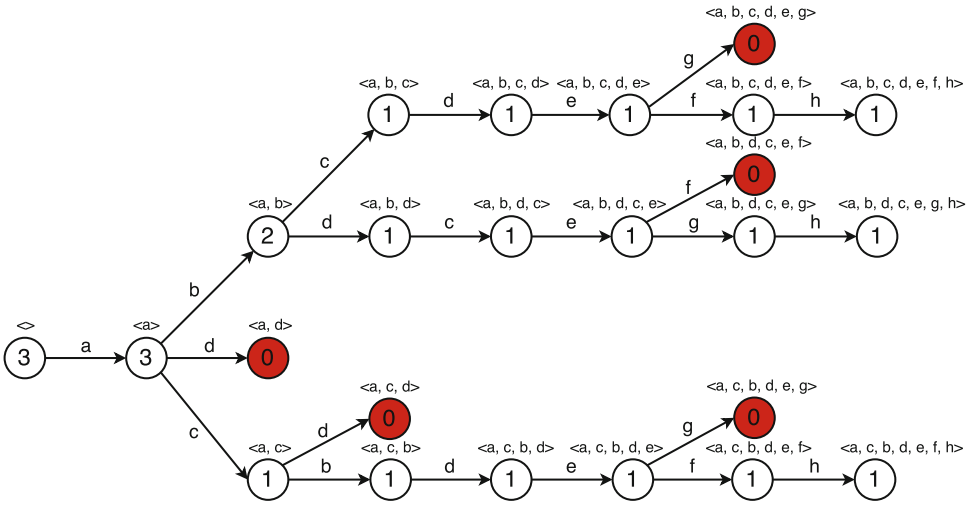
a	b	c	d	e	f	h
a	b	c	d	e	f	h

γ_6 =

a	c	b	≫	e	f	h
a	c	b	d	e	f	h

γ_7 =

a	b	d	c	≫	g	e	h
a	b	d	c	e	g	≫	h



Conformance Checking, Fig. 14 Prefix automaton $\mathcal{A}_1$ of alignments between log L_1 and model M_1 enhanced with model behavior

occur in the process. Assuming that the event log gives a complete view of the process (i.e., a log completeness assumption), the precision of the process model can be computed using artificial negative events and the concepts of precision and recall in data mining.

Artificial negative events can be induced by grouping similar traces and then observing the events that did not occur for every event in the traces. Under the log completeness assumption, this means that these unobserved events are negative events that are not allowed to happen by the process (Goedertier et al. 2009).

The process model can then be compared with the log by treating the model as a predictive model. For a given incomplete event sequence (i.e., an unfinished process instance), activities that are permitted by the model and observed in the log are classified as true positives (TP). Activities that are permitted by the model but are induced as negative events from the log are classified as false positive (FP). Activities that are not permitted by the model but observed in the log are classified as false negative (FN). Finally, activities that are not permitted by the model and are induced as negative events from the log are classified as true negative (TN). As shown in Fig. 15, precision and recall can be computed using a confusion matrix. Specifically, the precision of the positive class can be computed as:

$$precision = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The computed precision value corresponds to the precision of the three quality dimensions in process mining since it refers to the proportion of modeled behavior that is observed in the log.

The use of artificial negative events can also be extended to quantify generalization and to com-

	Actual Positive	Actual Negative
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

Conformance Checking, Fig. 15 Confusion matrix

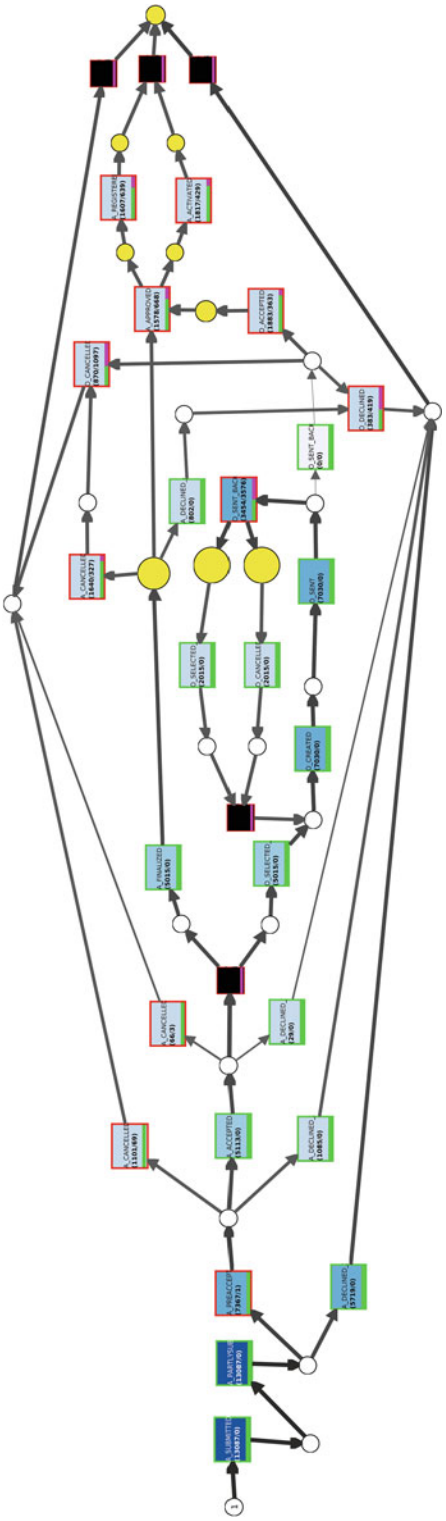
pute a precision metric that is more robust against less complete event logs. This is achieved by extending the artificial negative event induction strategy to assign weights to the induced negative events (vanden Broucke et al. 2014).

Examples of Application

All of the conformance checking techniques presented in the previous section have been implemented and are applicable to most real-life scenarios. In the following, the A* cost-based alignment technique is applied to a real-life dataset to illustrate how conformance checking can be applied to gain insights about a process. This example utilizes the data presented in the "Conformance Checking: What does your process do when you are not watching?" tutorial by de Leoni, van Dongen, and Munoz-Gama at the "15th International Conference on Business Process Management (BPM17)."

As previously presented, a process model and an event log are required to perform conformance checking. The real-life event log is taken from a Dutch financial institute and is of an application process for a personal loan or overdraft within a global financial organization (van Dongen 2012). This means that each case in the log records the occurred events of a particular loan application. The log contains some 262,200 events in 13,087 cases. Apart from some anonymization, the data is presented as it is recorded in the financial institute. The log is merged from three intertwined subprocesses so that the originating subprocess of each event can be identified by the first letter of the activity recorded by the event. In this example, the log is filtered so that it only contains events from two of the subprocesses: the process which records the state of the application (identifiable by "A_") and the process which records the state of an offer communicated to the applicant (identifiable by "O_"). The model has been created with the help of domain experts and can be assumed to be a realistic representation of the underlying process.

Figure 16 shows the process model projected with the computed alignment results to allow a visual diagnosis of the conformance results. For each transition, there is an error bar to show the distribution of synchronous moves (green) and model moves (pink) for the transition. For example, there are 383 synchronous moves and 419 model moves related to transition *O_DECLINED*. The occurrence and amount of log moves are indicated by highlighting places in yellow and the size of the highlighted places. Observing the model, one can note that transition *O_SENT_BACK* is associated with a large amount of model moves. This transition is quite an important part of the process as it corresponds to the event where the financial institute receives a reply from the applicant after a loan offer is made. A model move of *O_SENT_BACK* in a log trace means that the system did not register a reply from the applicant regarding a made offer as required by the process for the corresponding loan application. Investigation of cases with a model move in *O_SENT_BACK* (e.g., the case with caseId 174036) would show that there are cases for which an offer was created, sent, and accepted without having received a reply from the corresponding applicant. Whether it was due to a system error, an employee's mistake, or at worst a fraudulent case, clearly it is in the financial institute's best interest to investigate the root cause of this conformance issue.



Conformance Checking, Fig. 16 Process model projected with alignment results

Future Directions for Research

While there have been significant advances in the research of conformance checking over the recent years, there are still many open challenges and research opportunities. Some of them include:

Conformance Dimensions

The proposed three quality dimensions (fitness, precision, and generalization) have been widely accepted, but there is still a need for further understanding on how to interpret and quantify them through metrics. Furthermore, conformance can be extended beyond the current three dimensions, e.g., log completeness to quantify whether if the observed event data gives the full picture of the underlying process (Janssenswillen et al. 2017).

Big Data and Real Time

Process mining techniques and tools are getting applied to larger and more complex processes. This means that they have to be scalable to handle the increased size and complexity. In fact, much of the recent research efforts in conformance checking have been focused on this issue. Related research lines include decomposed conformance checking (Munoz-Gama 2016) and online conformance checking for event streams (Burattin 2015).

Conformance Diagnosis and Process Model Repair

It is not enough to just identify conformance issues; good diagnostic and visualization tools are crucial in helping the analyst identify and understand the root causes of the conformance issues. While there has been work done in this aspect of conformance checking, e.g., Buijs and Reijers (2014) and Munoz-Gama et al. (2014), there is much more to be done to provide better conformance diagnosis technology, e.g., new techniques and user study. Finally, once the differences between the model and the log have been diagnosed, the user may wish to repair the model in order to fix such differences and achieve a model that better describes the real process executed. This topic is extensively covered in the *Process Model Repair* entry of this encyclopedia.

Cross-References

- Automated Process Discovery
- Business Process Event Logs and Visualization
- Process Model Repair

References

Adriansyah A (2014) Aligning observed and modeled behavior. Ph.D. thesis, Eindhoven University of Technology

Adriansyah A, Munoz-Gama J, Carmona J, van Dongen BF, van der Aalst WMP (2012) Alignment based precision checking. In: Rosa ML, Soffer P (eds) Business process management workshops – BPM 2012 international workshops, Tallinn, Estonia, 3 Sept 2012. Revised papers. Lecture notes in business information processing, vol 132. Springer, pp 137–149

Buijs, JCAM, Reijers HA (2014) Comparing business process variants using models and event logs. In: Enterprise, business-process and information systems modeling – 15th international conference, BPMDS 2014, 19th international conference, EMMSAD 2014, held at CAiSE 2014, Thessaloniki, Greece, 6–17 June 1 2014. Proceedings, pp 154–168

Burattin A (2015) Process mining techniques in business environments – theoretical aspects, algorithms, techniques and open challenges in process mining. Lecture notes in business information processing, vol 207. Springer, Cham

Goedertier S, Martens D, Vanthienen J, Baesens B (2009) Robust process discovery with artificial negative events. J Mach Learn Res 10:1305–1340

Janssenswillen G, Donders N, Jouck T, Depaire B (2017) A comparative study of existing quality measures for process discovery. Inf Syst 71:1–15

Munoz-Gama J (2016) Conformance checking and diagnosis in process mining – comparing observed and modeled processes. Lecture notes in business information processing, vol 270. Springer, Cham

Munoz-Gama J, Carmona J, van der Aalst WMP (2014) Single-entry single-exit decomposed conformance checking. Inf Syst 46:102–122

Rozinat A, van der Aalst WMP (2008) Conformance checking of processes based on monitoring real behavior. Inf Syst 33(1):64–95

van der Aalst WMP (2013) Mediating between modeled and observed behavior: the quest for the "right" process: keynote. In: RCIS, pp 1–12. IEEE

van der Aalst WMP (2016) Process mining – data science in action. Springer, Berlin/Heidelberg

van der Aalst WMP, Adriansyah A, van Dongen BF (2012) Replaying history on process models for

conformance checking and performance analysis. Wiley Interdisc Rew Data Min Knowl Disc 2(2):182–192
van Dongen BF (2012) BPI challenge 2012. 4TU Datacentrum. Eindhoven University of Technology
vanden Broucke SKLM, Weerdt JD, Vanthienen J, Baesens B (2014) Determining process model precision and generalization with weighted artificial negative events. IEEE Trans Knowl Data Eng 26(8):1877–1889

Consistency Criterion

▸ Database Consistency Models

Consistency Model

▸ Database Consistency Models

Consistency Models in MMOGs

▸ Transactions in Massively Multiplayer Online Games

Constraint-Based Process Mining

▸ Declarative Process Mining

Context-Aware User Mobility

▸ Using Big Spatial Data for Planning User Mobility

Continuous Dataflow Language

▸ Stream Processing Languages and Abstractions

Continuous Queries

Martin Hirzel
IBM Research AI, Yorktown Heights, NY, USA

C

Synonyms

Streaming SQL queries; Stream-relational queries; StreamSQL queries

Definitions

A continuous query in an SQL-like language is a declarative query on data streams expressed in a query language for streams derived from the SQL for databases.

Overview

Just like data that is stored in a relational database can be queried with SQL, data that travels in a stream can be queried with an SQL-like query language. For databases, the relational model and its language, SQL, have been successful because the relational model is a foundation for clean and rigorous mathematical semantics and because SQL is declarative, specifying what the desired result is without specifying how to compute it (Garcia-Molina et al. 2008). However, the classic relational model assumes that data resides in relations in a database. When data travels in a stream, such as for communications, sensors, automated trading, etc., there is a need for continuous queries. SQL dialects for continuous queries fill this need and inherit the advantages of SQL and the relational model. Furthermore, SQL-like streaming languages capitalize on the familiarity of SQL for developers and of implementation techniques from relational databases.

There are various different SQL-like streaming languages. This article illustrates concepts using CQL (the continuous query language, Arasu et al. (2006)) as a representative example, because it has clean semantics and addresses the interplay between streams and relations.

Section "Findings" explores other SQL-like streaming languages beyond CQL.

SQL-Like Syntax

The surface syntax for streaming SQL dialects borrows familiar SQL clauses (such as select, from, and where) and augments them with streaming constructs (which turn streams into relations and vice versa). Consider the following CQL (Arasu et al. 2006) query with extensions adapted from Soulé et al. (2016):

```
1 Quotes  : { ticker : string, ask : int } stream;
2 History : { ticker : string, low : int } relation;
3 Bargains: { ticker : string, ask : int, low : int } stream
4   = select istream(*)
5     from Quotes[now], History
6     where Quotes.ticker == History.ticker and Quotes.ask <= History.low;
```

Line 1 declares *Quotes* as a stream of {*ticker*, *ask*} tuples, and Line 2 declares *History* as a relation of {*ticker*, *low*} tuples. Neither *Quotes* nor *History* is defined with a query in the example. Lines 3–6 declare *Bargains* and define it with a query. Line 3 declares *Bargains* as a stream of {*ticker*, *ask*, *low*} tuples. Line 4 specifies the output using the **istream** operator, which creates a stream from the insertions to a relation, using * to pick up all available tuple attributes. Line 5 joins two relations, *Quotes*[**now**] and *History*, where *Quotes*[**now**] creates a relation from the current contents of stream *Quotes*. Finally, Line 6 selects only tuples satisfying the given predicate, whereas any tuples for which the predicate is false are dropped.

The above example illustrates the core features of CQL, viz.: using operators such as **now** to turn streams into relations, using SQL to query relations, and using operators such as **istream** to turn relations into streams. For more detail, the following paragraphs explain the CQL grammar, starting with the top-level syntax:

$$
\begin{aligned}
&\textit{program} && ::= \textit{decl}^{+}\\
&\textit{decl} && ::= \textit{ID}\ \text{':'}\ \textit{tupleType}\ \textit{declKind}\ (\text{'='}\textit{query})^{?}\ \text{';'}\\
&\textit{tupleType} && ::= \text{'\{'}(\textit{ID}\ \text{':'}\ \textit{TYPE})^{+,}\ \text{'\}'}\\
&\textit{declKind} && ::= \text{'}\textbf{relation}\text{'}\,|\,\text{'}\textbf{stream}\text{'}\\
&\textit{query} && ::= \textit{select}\ \textit{from}\ \textit{where}^{?}\ \textit{groupBy}^{?}
\end{aligned}
$$

A program consists of one or more declarations. Each declaration has an identifier (*ID*), a tuple type (one or more attributes specified by their identifiers and types), a declaration kind (either **relation** or **stream**), and an optional query. The grammar meta-notation contains superscripts for optional items ($X^{?}$), repetition (X^{+}), and repetition separated by commas ($X^{+,}$). Finally, a query consists of mandatory select and from clauses and optional where and group-by clauses. Next, we look at the grammar for the select clause, which specifies the query output:

$$
\begin{aligned}
&\textit{select} && ::= \text{'}\textbf{select}\text{'}\ \textit{outputList}\ |\text{'}\textbf{select}\text{'}\ \textit{relToStream}\ \text{'('}\ \textit{outputList}\ \text{')'}\\
&\textit{relToStream} && ::= \text{'}\textbf{istream}\text{'}\ |\ \text{'}\textbf{dstream}\text{'}\ |\ \text{'}\textbf{rstream}\text{'}\\
&\textit{outputList} && ::= \text{'*'}\ |\ \textit{projectItem}^{+,}\ |\ \textit{aggrItem}^{+,}\\
&\textit{projectItem} && ::= \textit{expr}\ (\text{'}\textbf{as}\text{'}\ \textit{ID})^{?}\\
&\textit{aggrItem} && ::= \textit{AGGR}\ \text{'('}\textit{ID}^{*,}\ \text{')'}(\text{'}\textbf{as}\text{'}\ \textit{ID})^{?}
\end{aligned}
$$

A select clause either specifies an output list directly or wraps an output list in a relation-to-stream operator. In the first case, the query output is a relation, while in the second case, the query output is a stream. There are three relation-to-stream operators: **istream** captures insertions, **dstream** captures deletions, and **rstream** captures the entire relation at any given point in time. Here, a relation at a given point in time is a bag of tuples (i.e., an unordered collection of tuples that can contain duplicates). A stream is an unbounded bag of timestamped tuples (pairs of $\langle timestamp, tuple \rangle$). The grammar for *outputList* is borrowed from SQL. Next, we look at the grammar for the from clause, which specifies the query input:

C

from	::= '**from**' $inputItem^{+,}$
inputItem	::= *ID* ('[' *streamToRel* ']')$^{?}$ ('**as**' *ID*)$^{?}$
streamToRel	::= '**now**' | '**unbounded**' | *timeWindow* | *countWindow*
timeWindow	::= $partitionBy^{?}$ '**range**' *TIME* ('**slide**' *TIME*)$^{?}$
countWindow	::= $partitionBy^{?}$ '**rows**' *NAT* ('**slide**' *NAT*)$^{?}$
partitionBy	::= '**partition**' '**by**' $ID^{+,}$

An input item either identifies a relation directly or applies a stream-to-relation operator to a stream identifier. Stream-to-relation operators are written postfix in square brackets, reminiscent of indexing or slicing syntax in other programming languages. There are four such operators: **now** (tuples with the current timestamp), **unbounded** (tuples up to the current timestamp), **range** (time-based sliding window), and **rows** (count-based sliding window). Sliding windows can optionally be partitioned, in which case their size is determined separately and independently for each unique combination of the specified partitioning attributes. Sliding windows can optionally specify a slide granularity. Finally, we look at the grammar for *where* and *groupBy* as examples of other classic SQL clauses:

where	::= '**where**' *expr*
groupBy	::= '**group**' '**by**' $ID^{+,}$

The where clause selects tuples using a predicate expression, and the group-by clause specifies the scope for aggregation queries. These clauses in CQL are borrowed unchanged from SQL. For brevity, we omitted other classic SQL constructs, such as distinct, union, having, or other types of joins, which streaming SQL dialects such as CQL can borrow from SQL as is.

Typically, before execution, queries in SQL or its derivatives are first translated into a logical query plan of operators, which is the subject of the next section.

Stream-Relational Algebra

Whereas the SQL-like syntax of the previous section is designed to be authored by humans, this section describes an algebra designed to be optimized and executed by stream-query engines. The algebra is stream-relational because it augments with stream operators the relational algebra from databases. Relational algebra has well-understood semantics (Garcia-Molina et al. 2008), and the CQL authors rigorously defined the formal semantics for the additional streaming operators (Arasu and Widom 2004). The following paragraphs provide only an informal overview of the operators; interested readers can consult the literature for formal treatments. The notation for operator signatures is:

$$operator_{\langle \text{configuration} \rangle}(input) \rightarrow output$$

The configuration of an operator specializes its behavior. The input to an operator consists of one or multiple relations or a stream. And the output of an operator consists of either a relation or a stream. Relational algebra is compositional, since the output of one operator can be plugged

into the input of another operator, modulo compatible kind and type. For instance, the stream-relational algebra for stream *Bargains* in the CQL example from the start of section "SQL-Like Syntax" is:

$$\textbf{istream}\left(\sigma_{\langle ask \leq low \rangle}\left(\bowtie_{\langle Quotes.ticker = History.ticker \rangle} \left(\textbf{now}(Quotes), History\right)\right)\right)$$

Classic relational algebra has operators from relations to relations (Garcia-Molina et al. 2008):

- $\pi_{\langle assignments \rangle}(relation) \rightarrow relation$
  Project each input tuple using assignments to create a tuple in the output relation.
- $\sigma_{\langle predicate \rangle}(relation) \rightarrow relation$
  Select tuples for which the predicate is true, and filter out tuples for which it is false.
- $\bowtie_{\langle predicate \rangle}(relation^{+,}) \rightarrow relation$
  Join tuples from input relations as if with a cross product followed by $\sigma_{\langle predicate \rangle}$.
- $\gamma_{\langle groupBy, assignments \rangle}(relation) \rightarrow relation$
  Group tuples and then aggregate within each group, using the given assignments.

For brevity, we omitted other classic relational operators, but they could be added trivially, thanks to the compositionality of the algebra. CQL can use the well-defined semantics of classic relational algebra operators by applying them on snapshots of relations at a point in time. Some operators, such as π and σ, process each tuple in isolation, without carrying any state from one tuple to another (Xu et al. 2013). These operators could be easily lifted to work on streams, and indeed, some streaming SQL dialects do just that. But the same is not true for stateful operators such as $\bowtie$ and γ. To use these on streams, CQL first converts streams to relations, using window operators:

- **now**$(stream) \rightarrow relation$
  At each time instant t, all tuples from the input stream with timestamp exactly t.
- **unbounded**$(stream) \rightarrow relation$
  At each time instant t, all tuples from the input stream with timestamp at most t.
- **range**$_{\langle partitionBy, size(, slide)^? \rangle}(stream) \rightarrow relation$
  Use a time-based sliding window on the input stream as the output relation.
- **rows**$_{\langle partitionBy, size(, slide)^? \rangle}(stream) \rightarrow relation$
  Use a count-based sliding window on the input stream as the output relation.

These window operators correspond directly to the corresponding surface syntax discussed in section "SQL-Like Syntax". Gedik surveyed these and more window constructs and their implementation (Gedik 2013). A final set of operators turns relations back into streams:

- **istream**$(relation) \rightarrow stream$
  Watch input relation for insertions, and send those as tuples on the output stream.
- **dstream**$(relation) \rightarrow stream$
  Watch input relation for deletions, and send those as tuples on the output stream.
- **rstream**$(relation) \rightarrow stream$
  At each time instant, send all tuples currently in input relation on output stream.

This article illustrated continuous queries in SQL-like languages using CQL as the concrete example, because it is clean and well studied. The original CQL work contains more details and a denotational semantics (Arasu et al. 2006; Arasu and Widom 2004). Soulé et al. furnish CQL with a static type system and formalize translations from CQL via stream-relational algebra to a calculus with an operational semantics (Soulé et al. 2016).

Findings

CQL was not the first dialect of SQL for streaming. TelegraphCQ reused the front-end of PostgreSQL as the basis for its surface language (Chandrasekaran et al. 2003). Rather than focusing on surface language innovation, TelegraphCQ focused on a stream-relational algebra back-end that pioneered new techniques for dynamic optimization and query sharing. Gigascope had its own dialect of SQL called GSQL (Cranor et al. 2003). Unlike CQL, GSQL used an algebra where all operators work directly on streams. As discussed earlier, this is straightforward for π and σ, but not for $\bowtie$ and γ. Therefore, GSQL required $\bowtie$ and γ to be configured with constraints over ordering attributes that effectively function as windows. Aurora used a graphical interface for surface-level programming, but we still consider it an SQL-like system, because it used a stream-relational algebra (Abadi et al. 2003). Aurora's Stream Query Algebra (SQuAl) contained the usual operators π, σ, $\bowtie$, and γ, as well as union, sort, and a resample operator that interpolates missing values.

CQL took a more language-centric approach than its predecessors. It also inspired work probing semantic subtleties in SQL-like streaming languages. Jain et al. precisely define the semantics for the corner case of StreamSQL behavior where multiple tuples have the same timestamp (Jain et al. 2008). In that case, there is no inherent order among these tuples, so tuple-based windows must choose arbitrarily, leading to undefined results. Furthermore, if actions are triggered on a per-tuple basis, there can be multiple actions at a single timestamp, leading to spurious intermediate results that some would consider a glitch. The SECRET paper is also concerned with problems of time-based sliding windows (Botan et al. 2010). SECRET stands for ScopE (which timestamps belong to a window), Contents (which tuples belong to a window), REport (when to output results), and Tick (when to trigger computation). Finally, Zou et al. explored turning repeated SQL queries into continuous CQL queries by turning parameters that change between successive invocations into an input stream (Zou et al. 2010).

Today, there is still much active research on big-data streaming systems, but the focus has shifted from SQL dialects to embedded domain-specific languages (EDSLs, Hudak 1998). An EDSL for streaming is a library in a host language that offers abstractions for continuous queries. In practice, most EDSLs lack the rigorous semantics of stand-alone languages such as CQL but have the advantage of posing a lower barrier to entry for developers who are already proficient in the host language, being easier to extend with user-defined operators, and not requiring a separate language tool-chain (compiler, debugger, integrated development environment, etc.).

Examples

The beginnings of sections "SQL-Like Syntax" and "Stream-Relational Algebra" show a concrete example of the same query in CQL and in stream-relational algebra, respectively. The most famous example of a set of continuous queries written in an SQL-like language is the Linear Road benchmark. The benchmark consists of computing variable-rate tolling for congestion pricing on highways. A simplified version of Linear Road serves to motivate and illustrate CQL (Arasu et al. 2006). The full version of Linear Road is presented in a paper of its own (Arasu et al. 2004). Both the simplified version and the full version of the benchmark continue to be popular, and not just for SQL-inspired streaming systems and languages (Grover et al. 2016; Hirzel et al. 2016; Jain et al. 2006; Soulé et al. 2016).

Future Directions for Research

Stream processing is an active area of research, and new papers often use a streaming SQL foundation to present their results. One challenging issue for streaming systems is how to handle

out-of-order data. For instance, CEDR suggests a solution based on stream-relational algebra using several timestamps per tuple (Barga et al. 2007). One challenge that is particular to SQL-like languages is how to extend them with user-defined operators without losing the well-definedness of the restricted algebra. For instance, StreamInsight addresses this issue with an extensibility framework (Ali et al. 2011). Finally, the semantics of SQL are defined by reevaluating relational operators on windows whenever the window contents change. Nobody suggests that this reevaluation is the most efficient approach, but developing better solutions is an interesting research challenge. For instance, the DABA algorithm performs associative sliding-window aggregation on FIFO windows in worst-case constant time (Tangwongsan et al. 2017). All three of the abovementioned research topics (out-of-order processing, extensibility, and incremental streaming algorithms) are still ripe with open issues.

Cross-References

▸ Stream Processing Languages and Abstractions

References

Abadi DJ, Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S (2003) Aurora: a new model and architecture for data stream management. J Very Large Data Bases (VLDB J) 12(2):120–139

Ali M, Chandramouli B, Goldstein J, Schindlauer R (2011) The extensibility framework in Microsoft StreamInsight. In: International conference on data engineering (ICDE), pp 1242–1253

Arasu A, Widom J (2004) A denotational semantics for continuous queries over streams and relations. SIGMOD Rec 33(3):6

Arasu A, Cherniack M, Galvez E, Maier D, Maskey AS, Ryvkina E, Stonebraker M, Tibbetts R (2004) Linear road: a stream data management benchmark. In: Conference on very large data bases (VLDB), pp 480–491

Arasu A, Babu S, Widom J (2006) The CQL continuous query language: semantic foundations and query execution. J Very Large Data Bases (VLDB J) 15(2): 121–142

Barga RS, Goldstein J, Ali M, Hong M (2007) Consistent streaming through time: a vision for event stream processing. In: Conference on innovative data systems research (CIDR), pp 363–373

Botan I, Derakhshan R, Dindar N, Haas L, Miller RJ, Tatbul N (2010) SECRET: a model for analysis of the execution semantics of stream processing systems. In: Conference on very large data bases (VLDB), pp 232–243

Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden S, Raman V, Reiss F, Shah MA (2003) TelegraphCQ: continuous dataflow processing for an uncertain world. In: Conference on innovative data systems research (CIDR)

Cranor C, Johnson T, Spataschek O, Shkapenyuk V (2003) Gigascope: a stream database for network applications. In: International conference on management of data (SIGMOD) industrial track, pp 647–651

Garcia-Molina H, Ullman JD, Widom J (2008) Database systems: the complete book, 2nd edn. Pearson/Prentice Hall, London, UK

Gedik B (2013) Generic windowing support for extensible stream processing systems. Softw Pract Exp (SP&E) 44:1105–1128

Grover M, Rea R, Spicer M (2016) Walmart & IBM revisit the linear road benchmark. https://www.slideshare.net/RedisLabs/walmart-ibm-revisit-the-linear-road-benchmark (Retrieved Feb 2018)

Hirzel M, Rabbah R, Suter P, Tardieu O, Vaziri M (2016) Spreadsheets for stream processing with unbounded windows and partitions. In: Conference on distributed event-based systems (DEBS), pp 49–60

Hudak P (1998) Modular domain specific languages and tools. In: International conference on software reuse (ICSR), pp 134–142

Jain N, Amini L, Andrade H, King R, Park Y, Selo P, Venkatramani C (2006) Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: International conference on management of data (SIGMOD), pp 431–442

Jain N, Mishra S, Srinivasan A, Gehrke J, Widom J, Balakrishnan H, Cetintemel U, Cherniack M, Tibbets R, Zdonik S (2008) Towards a streaming SQL standard. In: Conference on very large data bases (VLDB), pp 1379–1390

Soulé R, Hirzel M, Gedik B, Grimm R (2016) River: an intermediate language for stream processing. Softw Pract Exp (SP&E) 46(7):891–929

Tangwongsan K, Hirzel M, Schneider S (2017) Low-latency sliding-window aggregation in worst-case constant time. In: Conference on distributed event-based systems (DEBS), pp 66–77

Xu Z, Hirzel M, Rothermel G, Wu KL (2013) Testing properties of dataflow program operators. In: Conference on automated software engineering (ASE), pp 103–113

Zou Q, Wang H, Soulé R, Hirzel M, Andrade H, Gedik B, Wu KL (2010) From a stream of relational queries to distributed stream processing. In: Conference on very large data bases (VLDB) industrial track, pp 1394–1405

Continuous Query Optimization

▸ Stream Query Optimization

Coordination Avoidance

Faisal Nawab
University of California, Santa Cruz, CA, USA

Definitions

Coordination avoidance denotes a class of distributed system methods that minimize the amount of coordination between nodes while maintaining the integrity of the application.

Overview

In many data management systems, data and processing are replicated or distributed across nodes (Kemme et al. 2010; Bernstein and Goodman 1981). This replication and distribution increase the levels of fault tolerance and availability. However, they introduce a coordination cost to maintain the integrity of applications. Since nodes are processing data for the same application simultaneously, there is the possibility of conflicting operations that may overwrite the work of other nodes. To overcome this problem, coordination and synchronization protocols have been developed to ensure the integrity of data. Typically, the coordination protocols strive to ensure a guarantee of correctness, such as serializability (Bernstein et al. 1987) and linearizability (Herlihy and Wing 1990). These are strong notions of correctness that model the correctness of a distributed system by whether the data management system can mask the underlying concurrency from the upper layers. Serializability, for example, guarantees that the outcome of the execution of distributed transactions (i.e., bundles of operations) is equivalent to some serial execution. Thus, the application developer has the illusion of the existence of a single machine that executes transactions serially.

A strong guarantee of correctness is extremely useful for application developers because it frees them from thinking about concurrency and replication issues. Additionally, relying on the application level to deal with concurrency anomalies is error-prone and may lead to many wasted efforts in applications reinventing the wheel. Rather, the data management approach separates concerns and relies on a data management layer to manage concurrency and replication to guarantee the consistency of the application. However, these approaches are expensive, requiring extensive coordination between nodes, as any two operations touching the same data item need to be synchronized. Data management systems tackled this problem by building better coordination and synchronization designs and protocols to optimize performance. Unfortunately, this continues to be a challenging task as distributed systems are becoming bigger with large-scale many-node and many-core deployments that increase the coordination and communication demands. Also, distributed systems are increasingly being deployed across wide geographical locations, increasing the latency of communication. These two trends continue to be adopted and are made accessible through cloud technology. To reduce the cost of coordination, there have been approaches that relax the consistency guarantees and trade them off with better performance. This approach, however, does not retain the easy-to-use data management abstractions that can be provided with strong consistency guarantees.

Coordination avoidance aims to reduce the cost of coordination while maintaining the integrity of data and the easy-to-use data management abstractions. The approach of coordination avoidance exploits knowledge of the application layer to extract the set of consistency guarantees that are sufficient for its application-level correctness. Therefore, the cost of coordination is only incurred when there is a potential that it will lead to an anomaly. Extensions to coordination avoidance approaches build execution

models and abstractions that facilitate and enable avoiding coordination.

Key Research Findings

The goal of coordination avoidance techniques and protocols is to minimize the cost of coordination while retaining the application integrity. There have been a plethora of work that investigates such approaches. What these works have in common is attempting to leverage application-level semantics to deduce the set of guarantees that are sufficient for a consistent execution. They may take various forms that are presented in this section. Coordination avoidance techniques combine a subset of different approaches. Generally, a coordination avoidance protocol tackle two tasks: extracting or modeling a set of consistency semantics for a particular application and executing operations while minimizing coordination to the cases when only the set of extracted consistency semantics are threatened.

There are various approaches for extracting consistency semantics that vary in their ease of use, generality, and optimality, where coordination avoidance protocols typically face the trade-off between these three properties. One approach extracts consistency semantics from existing code via the use of static analysis or other similar methods. This approach requires the least intervention from application developers. However, it introduces the challenge of extracting features from general code which can be limited in its scope. Some other approaches rely on the application developers to annotate or model their code to enable extracting the consistency semantics. This approach requires additional intervention from the application developer, although allowing the use of existing general code. Alternatively, some works explore an approach where specialized frameworks and abstractions are introduced to facilitate and enable the extraction of consistency semantics. With the extracted features, there are various approaches to execute distributed applications that also vary in their ease of use, generality, and optimality. One approach is to selectively coordinate between nodes according to the extracted consistency semantics. Another approach is to provide specialized protocols and execution mechanisms that guarantee the consistency semantics. The following are examples of coordination avoidance methods and work that combine and explore different extraction and execution approaches.

Commutativity and Convergence

A large number of coordination avoidance protocols leverage the commutativity of operations. Commutative operations are ones that can be reordered arbitrarily while reaching the same final state. Therefore, commutative operations can execute at different nodes in different orders without synchronously coordinating with other nodes. The operations are asynchronously propagated to other nodes that execute them without the need of any ordering. Commutative operations are especially useful for hot spots that are frequently being updated, such as counters. Many attempts have been made to generalize the notion of commutativity in the context of data management (Badrinath and Ramamritham 1992; Korth 1983). Recoverability (Badrinath and Ramamritham 1992) generalizes commutativity to include more operations by explicitly ordering commit points. For example, two operations pushing to a stack simultaneously are not commutative. However, with recoverability, they can execute concurrently while maintaining serializability with the condition that they commit in the order they are invoked. Korth (1983) generalizes locking to account for the existence of commutative operations and thus allow more concurrency when commutative operations are being used. Commutative operations can also be used in conjunction with other consistency guarantees. Walter (Sovran et al. 2011), for example, is a system that employs a consistency notion based on snapshot isolation. Walter allows commutative operations to be performed without coordination.

Convergent and commutative replicated data types (CRDTs) (Shapiro et al. 2011) are an approach to make the use of commutativity more powerful by designing access abstractions and data structures that are more general than

traditional commutative operations while still maintaining commutativity. This has led to the development of nontrivial data structures that perform complex operations while being commutative, thus requiring no coordination (Preguica et al. 2009). CRDTs are also used in the context of eventual consistency by providing a means to converge operations to the same, eventually consistent state. Also, it is used in coordination-free protocols to converge operations that do not need to be synchronously coordinated (Bailis et al. 2014).

CALM (Alvaro et al. 2011) is a principle that shows that a distributed program can be executed without coordination if the program's logic is monotone. This enables avoiding coordination by building programs that ensure logical monotonicity in addition to using analysis techniques to assist whether a given program is logically monotonic. Bloom is a domain-specific declarative language that facilitates leveraging the CALM principle to build coordination-free programs. In this approach, the whole program does not have to be monotonic, rather, it is possible to detect potential anomalies that need to be addressed by the developer.

Application-Level Correctness Semantics

Many data management system works utilize application-level invariants and correctness semantics to avoid coordination (Lamport 1976; Agrawal et al. 1993, 1994; Garcia-Molina 1983; Li et al. 2012; Roy et al. 2015; Bailis et al. 2014). In this approach, operations are allowed to execute and commit without coordination if the application-level correctness semantics are not violated. Lamport (1976) (as described in Agrawal et al. 1993) is the first to show that application-level semantics can be used to avoid coordination between certain types of transactions. Garcia-Molina (1983) introduced a design of a concurrency control protocol that leverages application-level correctness semantics. The protocol receives application-level semantics from users and then uses them to allow nonserializable schedules that do not violate the application's correctness semantics. To achieve this, the protocol divides the transaction into steps and derives what steps can interleave with each other. Other works adopt a similar approach of dividing the transaction to find interleaving between the parts of the transaction. A set of work has shown that performing this division hierarchically may lead better results (Lynch 1983; Weikum 1985; Farrag and Özsu 1989). A formalization of the correctness of application-level consistency definitions is provided in Korth and Speegle (1988).

Defining how the application-level consistency semantics are expressed is important for the practicality of the solution (i.e., how accessible is the interface to developers) and the performance gains of the solution (i.e., how the interface can maximize the amount of captured knowledge). Agrawal et al. (1993) propose the use of special operations, called consistency assertions, that captures a user's view of the database and the criterion for correct concurrency execution. Two correctness criteria are developed to leverage the special operations by making the database allow concurrent execution of two transactions if the consistency assertions are not violated. Relative serializability (Agrawal et al. 1994) extends the transaction model by allowing users to define the atomicity requirements of transactions relative to each other. Then, it is possible to interleave parts of transactions that do not violate the defined atomicity requirements.

The use of application-level correctness invariants is especially important for large-scale and geo-scale deployments where the coordination cost is high. This has motivated the development of more solutions that use application-level correctness invariants for these new architectures (Li et al. 2012; Roy et al. 2015; Bailis et al. 2014; Zhang et al. 2013). Gemini (Li et al. 2012) enables the coexistence of fast, eventually consistent operations with strongly consistent, slow operations. Therefore, operations are executed with high performance, and the cost of enforcing consistency is only paid when necessary. To facilitate this, a consistency model, called RedBlue consistency, is proposed. Using the consistency formulation and application-level consistency invariants, operations are assigned to

be either fast or consistent. The assignment is made with the guarantee that executing fast operations without synchronous coordination would not lead to violations of the application-level consistency semantics. Specifically, fast operations execute locally at one of the data centers and then lazily propagate to other data centers. Fast operations are only guaranteed to be causally ordered. Consistent operations, on the other hand, are synchronously synchronized, potentially with other data centers, leading to high latency. To be either the Homeostasis protocol (Roy et al. 2015) aims at automating the process of generating application-level correctness invariants. A program analysis technique is used to extract the application-level consistency invariants from code. In the Homeostasis protocol, the target consistency level is serializability, and the code is analyzed to extract the invariants that will lead to a serializable execution as observed by users. Application developers can then refine the extracted application-level correctness invariants. The Homeostasis protocol may enable and inspire future work on the use of automatic extraction of correctness invariants that improve their accuracy. However, it is also useful as a tool to bootstrap the process of specifying application-level correctness invariants.

Transaction chains (Zhang et al. 2013) leverage transaction chopping (Shasha et al. 1995) to commit transactions in a geo-scale deployment locally without waiting for other data centers. Transaction chopping (Shasha et al. 1995) is a theoretical framework that enables decomposing transactions to smaller pieces. The original goal of transaction chopping is to allow more concurrency. Transaction chains (Zhang et al. 2013) use the concepts of transaction chopping to decompose transaction. Then, it introduces constraints on the execution of transactions. Transaction parts are executed in the same order at the different nodes (potentially at different data centers). Also, instances of the same transaction are ordered at the first node that handles that type of transactions. It turns out that these two constraints, in addition to transaction chopping, allow guaranteeing the success of the execution of the transaction after the success in the first node. Thus, rather than waiting for the response from multiple nodes at multiple data centers, transaction can execute and be guaranteed to succeed after execution at the first node only. Transaction chains incorporate many annotation and commutative techniques to enable more flexibility in the schedules and constraints that are enforced on transactions.

Weak and Relaxed Consistency

Another method to avoid coordination is to weaken or relax the consistency guarantees. Eventual consistency (Cooper et al. 2008; DeCandia et al. 2007; Bailis and Ghodsi 2013) only guarantees that the data copies are eventually consistent in the absence of updates. Therefore, coordination can be performed asynchronously, and convergence can be achieved via commutative and converging operators. Causality (Lamport 1978) is a stronger guarantee that ensures that events at one node are totally ordered and events at one node are ordered after everything that have been received by the point they are generated. Causality does not require synchronous coordination, which made it a candidate for many environments with high coordination costs (Lloyd et al. 2011, 2013; Bailis et al. 2013; Nawab et al. 2015; Du et al. 2013). Extensions of causality that consider causal transactions rather than operations or events are also shown to require no synchronous coordination (Lloyd et al. 2011, 2013; Nawab et al. 2015). Parallel snapshot isolation (PSI) (Sovran et al. 2011) extends snapshot isolation (Berenson et al. 1995) for geo-scale environments. This extension allows many types of transactions to execute without the need for synchronous coordination between data centers. Transactions with write-write conflicts, however, are still required to synchronously coordinate.

Versioning and Snapshots

Serializability is a guarantee that the outcome of transactions mimics an outcome of a serial execution. This makes read-only transactions have the opportunity to execute without coordination by reading from a consistent snapshot of data.

The outcome of such read-only transaction might be stale but consistent. However, the challenge in read-only transactions is in the development of methods that minimize the interference with read-write transactions. This is especially the case for analytics queries that translate to large, long-lived read-only transactions. Schlageter (1981) showed that read-only transactions can be executed without interfering with read-write transactions. Specifically, in an optimistic concurrency control protocol, read-only transactions do not need to be validated as long as they reflect the consistent state of the database. However, read-write transactions wait for read-only transactions to finish. This may lead to long delays with large read-only transactions. To overcome this limitation, multi-versioning techniques were proposed to execute read-only transactions on a recent, possibly stale snapshot of the database (Agrawal et al. 1987). In a replicated system, it was shown that a read-only transaction can avoid coordination with other replicas by reading a local, consistent copy (Satyanarayanan and Agrawal 1993). This entails modifying the coordination of read-write transactions and additional overhead to enable creating a consistent snapshot locally at every replica. Recently, Spanner (Corbett et al. 2012) shows the use of accurate time synchronization to serve consistent read-only transactions with improved freshness.

Conclusion

Coordination is continuing to be a major challenge in data management systems due to large-scale and geo-scale deployments. Coordination avoidance techniques enable reducing the overhead of coordination, thus improving performance, while maintaining the ease of use and intuition of data management abstractions and strong consistency guarantees. There have been many ways that can be used individually or together to avoid coordination such as leveraging commutativity and convergence, application-level consistency semantics, relaxed consistency guarantees, and specialized transaction types. These solutions provide different levels of ease of use, generality, and optimality, making them suitable for different classes of applications and use cases.

Examples of Application

Coordination avoidance is used in traditional and cloud data management systems.

Future Directions for Research

The emergence of many new technologies, system architectures, and applications invites the investigation of new coordination avoidance techniques and adaptations of existing coordination avoidance technique.

Cross-References

- Geo-Replication Models
- Geo-Scale Transaction Processing
- Weaker Consistency Models/Eventual Consistency

References

Agrawal D, Bernstein AJ, Gupta P, Sengupta S (1987) Distributed optimistic concurrency control with reduced rollback. Distrib Comput 2(1):45–59. https://doi.org/10.1007/BF01786254

Agrawal D, El Abbadi A, Singh AK (1993) Consistency and orderability: semantics-based correctness criteria for databases. ACM Trans Database Syst (TODS) 18(3):460–486

Agrawal D, Bruno JL, El Abbadi A, Krishnaswamy V (1994) Relative serializability (extended abstract): an approach for relaxing the atomicity of transactions. In: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. ACM, pp 139–149

Alvaro P, Conway N, Hellerstein JM, Marczak WR (2011) Consistency analysis in bloom: a calm and collected approach. In: CIDR, pp 249–260

Badrinath B, Ramamritham K (1992) Semantics-based concurrency control: beyond commutativity. ACM Trans Database Syst (TODS) 17(1):163–199

Bailis P, Ghodsi A (2013) Eventual consistency today: limitations, extensions, and beyond. Queue

11(3):20:20–20:32. https://doi.org/10.1145/2460276.2462076

Bailis P, Ghodsi A, Hellerstein JM, Stoica I (2013) Bolt-on causal consistency. In: Proceedings of the 2013 ACM SIGMOD international conference on management of data, SIGMOD'13. ACM, New York, pp 761–772. https://doi.org/10.1145/2463676.2465279

Bailis P, Fekete A, Franklin MJ, Ghodsi A, Hellerstein JM, Stoica I (2014) Coordination avoidance in database systems. Proc VLDB Endow 8(3):185–196

Berenson H, Bernstein P, Gray J, Melton J, O'Neil E, O'Neil P (1995) A critique of ansi SQL isolation levels. ACM SIGMOD Rec 24:1–10. https://doi.org/10.1145/223784.223785

Bernstein PA, Goodman N (1981) Concurrency control in distributed database systems. ACM Comput Surv (CSUR) 13(2):185–221

Bernstein PA, Hadzilacos V, Goodman N (1987) Concurrency control and recovery in database systems. Addison-Wesley, Reading

Cooper BF, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen HA, Puz N, Weaver D, Yerneni R (2008) Pnuts: Yahoo!'s hosted data serving platform. Proc VLDB Endow 1(2):1277–1288. https://doi.org/10.14778/1454159.1454167

Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, Hsieh W, Kanthak S, Kogan E, Li H, Lloyd A, Melnik S, Mwaura D, Nagle D, Quinlan S, Rao R, Rolig L, Saito Y, Szymaniak M, Taylor C, Wang R, Woodford D (2012) Spanner: Google's globally-distributed database, pp 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905

DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W (2007) Dynamo: Amazon's highly available key-value store. In: Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles, SOSP'07. ACM, New York, pp 205–220. https://doi.org/10.1145/1294261.1294281

Du J, Elnikety S, Roy A, Zwaenepoel W (2013) Orbe: scalable causal consistency using dependency matrices and physical clocks. In: Proceedings of the 4th annual symposium on cloud computing, SOCC'13. ACM, New York, pp 11:1–11:14. https://doi.org/10.1145/2523616.2523628

Farrag AA, Özsu MT (1989) Using semantic knowledge of transactions to increase concurrency. ACM Trans Database Syst (TODS) 14(4):503–525

Garcia-Molina H (1983) Using semantic knowledge for transaction processing in a distributed database. ACM Trans Database Syst (TODS) 8(2):186–213

Herlihy MP, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3):463–492. https://doi.org/10.1145/78969.78972

Kemme B, Jimenez-Peris R, Patino-Martinez M (2010) Database replication. Synth Lect Data Manag 2(1):1–153. http://www.morganclaypool.com/doi/abs/10.2200/S00296ED1V01Y201008DTM007

Korth HF (1983) Locking primitives in a database system. J ACM 30(1):55–79. https://doi.org/10.1145/322358.322363

Korth HK, Speegle G (1988) Formal model of correctness without serializabilty. ACM SIGMOD Rec 17: 379–386

Lamport L (1976) Towards a theory of correctness for multi-user data base system. Teeh. Rep. Tech. Rep., TRCA-7610-0712, Massachusetts Computer Associates

Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565. https://doi.org/10.1145/359545.359563

Li C, Porto D, Clement A, Gehrke J, Preguiça NM, Rodrigues R (2012) Making geo-replicated systems fast as possible, consistent when necessary. In: OSDI, vol 12, pp 265–278

Lloyd W, Freedman MJ, Kaminsky M, Andersen DG (2011) Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: Proceedings of the twenty-third ACM symposium on operating systems principles, SOSP'11. ACM, New York, pp 401–416. https://doi.org/10.1145/2043556.2043593

Lloyd W, Freedman MJ, Kaminsky M, Andersen DG (2013) Stronger semantics for low-latency geo-replicated storage. In: Proceedings of the 10th USENIX conference on networked systems design and implementation, NSDI'13. USENIX Association, Berkeley, pp 313–328. http://dl.acm.org/citation.cfm?id=2482626.2482657

Lynch NA (1983) Multilevel atomicity: a new correctness criterion for database concurrency control. ACM Trans Database Syst (TODS) 8(4):484–502

Nawab F, Arora V, Agrawal D, El Abbadi A (2015) Chariots: a scalable shared log for data management in multi-datacenter cloud environments. In: Proceedings of the 18th international conference on extending database technology, EDBT 2015, Brussels, pp 13–24. https://doi.org/10.5441/002/edbt.2015.03

Preguica N, Marques JM, Shapiro M, Letia M (2009) A commutative replicated data type for cooperative editing. In: 29th IEEE international conference on distributed computing systems, ICDCS'09. IEEE, pp 395–403

Roy S, Kot L, Bender G, Ding B, Hojjat H, Koch C, Foster N, Gehrke J (2015) The homeostasis protocol: avoiding transaction coordination through program analysis. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data. ACM, pp 1311–1326

Satyanarayanan OT, Agrawal D (1993) Efficient execution of read-only transactions in replicated multiversion databases. IEEE Trans Knowl Data Eng 5(5):859–871. https://doi.org/10.1109/69.243514

Schlageter G (1981) Optimistic methods for concurrency control in distributed database systems. In: Proceedings of the seventh international conference on very large data bases, vol 7, VLDB Endowment, VLDB'81, pp 125–130. http://dl.acm.org/citation.cfm?id=1286831.1286844

Shapiro M, Preguiça N, Baquero C, Zawirski M (2011) Convergent and commutative replicated data types. Bull Eur Assoc Theor Comput Sci (104):67–88

Shasha D, Llirbat F, Simon E, Valduriez P (1995) Transaction chopping: algorithms and performance studies. ACM Trans Database Syst (TODS) 20(3):325–363

Sovran Y, Power R, Aguilera MK, Li J (2011) Transactional storage for geo-replicated systems. In: Proceedings of the twenty-third ACM symposium on operating systems principles, SOSP'11. ACM, New York, pp 385–400. https://doi.org/10.1145/2043556.2043592

Weikum G (1985) A theoretical foundation of multi-level concurrency control. In: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on principles of database systems. ACM, pp 31–43

Zhang Y, Power R, Zhou S, Sovran Y, Aguilera MK, Li J (2013) Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles, SOSP'13. ACM, New York, pp 276–291. https://doi.org/10.1145/2517349.2522729

Co-resident Attack in Cloud Computing: An Overview

Sampa Sahoo, Sambit Kumar Mishra, Bibhudatta Sahoo, and Ashok Kumar Turuk
National Institute of Technology Rourkela, Rourkela, India

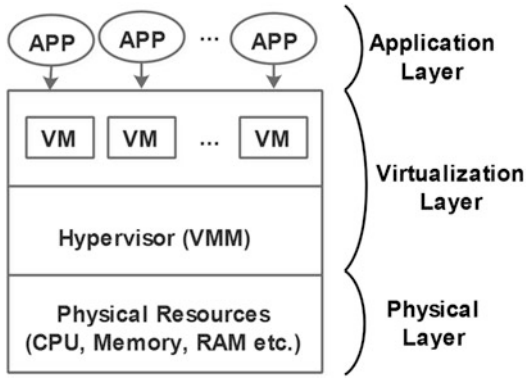
Introduction

Cloud computing is a novel computing model that leverages on-demand provisioning of computational and storage resources. It uses pay-as-you-go economic model, which helps to reduce CAPEX and OPEX. Cloud computing essence and scope can be best described by the "5-3-2 principle," where 5 stands for its essential characteristics (on-demand service, ubiquitous network access, resource pooling, rapid elasticity, and measured service) (Puthal et al. 2015). Three stands for service delivery models (SaaS, PaaS, and IaaS). Two points to basic deployment models (private and public) of the cloud (Sahoo et al. 2015).

Figure 1 shows the layered architecture of cloud. Various layers of a cloud are as follows: application, virtualization, and physical. Application layer consists of user applications. The virtualization layer's main components are a hypervisor or Virtual Machine Manager (VMM) and VM. A hypervisor acts as an administrator for the virtualized data center that permits the user to configure and manage cloud resources to create and deploy virtual machines and services in a cloud. A VM is an emulation of a physical machine. The virtualization technology allows creating several VMs on a single physical machine (Mishra et al. 2018a). The physical layer is a container of various physical resources like memory, RAM, etc. (Sahoo et al. 2016).

The rise of multimedia data and ever-growing requests for multimedia applications has tendered multimedia as the biggest contributor of big data. Applications on entertainment demand low security as compared to personalized videos like business meetings, telemedicine, etc. A secured video transmission ensures authorized entrance and prevents information leakage by any unapproved eavesdroppers. Various cryptographic algorithms (AES, DES, etc.) can be used to guarantee security to the multimedia application. Few examples of big data are credit card transactions, Walmart customer transactions, and data generated by Facebook users. A security breach in big data will cause severe legal consequences and reputational damage than at present. The use of big data in business also gives rise to privacy issues (Inukollu et al. 2014). Big data applications demand scalable and cost-efficient technology for huge data storage, computation, and communication. One of the solutions to address these issues is the use of cloud computing which is both scalable and cost-efficient. Since cloud computing is an umbrella term used by many technologies like networks, databases, virtualization, operating system, resource scheduling, transmission management, load balancing, etc., security issues related to these systems are also applicable to cloud computing.

Cloud computing facilitates business models and allows renting of IT resources on a utility like a basis through the Internet. Security is one of the most challenging issues to deal with in cloud computing due to numerous forms of



Co-resident Attack in Cloud Computing: An Overview, Fig. 1 Layered architecture of cloud

attacks on the application side as well as the hardware components. Various reasons for difficulties correlated with cloud computing are the loss of control, lack of trust, and multi-tenancy (Bhargava 2010). A cloud user outsourced its content to a remote server, losing direct control over the data, which becomes one of the reasons for cloud security issues. An attacker can target the communication link between cloud provider and clients. Multi-tenants of cloud use a shared pool of resources having conflicting goals. This gives opportunities to an attacker to steal useful information from the legitimate user on the same physical machine.

The virtualization method allows sharing of computing resources among many tenants, which may be business partners, suppliers, competitors, or attackers (Mishra et al. 2018b). Even though there is substantial logical isolation between VMs, shared hardware creates vulnerabilities to side channel attacks, i.e., data leakage through implicit channels (Zhang et al. 2011). The following sections discuss security issues in the cloud, co-resident attack, and its defense methods, followed by the conclusion.

Security Issues in Cloud

Security is a dominant barrier to the development and widespread use of cloud computing. Big data computing in cloud environment further intensified security and privacy issues. The cheaper compute environment, networked application environment, and shared cloud system environment generate security, compression, encryption, and access control challenges, which must be addressed in a systematic way (Cloud-Security-Alliance 2012). Cloud provider security implementation mainly focused on the assumption that outsiders of the cloud are evil, whereas insiders are good. An outside attacker performs the following actions to get unauthorized access: listen to network traffic, insert malicious traffic, and launch DoS. An inside attacker can log on to user communication, peek into VMs or make copies of VMs, and monitor application pattern for unapproved access. The deficit security features in local devices also provide a way for malicious services on the cloud, as an attacker can use these devices to attack local networks.

Various terms representing the security or privacy attributes are confidentiality, integrity, availability, accountability, and privacy preservability (Xiao and Xiao 2013). Confidentiality ensures that user's data and computation are kept confidential from both the cloud provider and other users. Primary threats to confidentiality are co-resident VM attack and malicious system admin. Cloud integrity implies that data are stored honestly on data storage, computation is performed without any distortion, and integrity is affected by data loss or manipulation and dishonest computation. Availability ensures that cloud services are available most of the time and meet the SLA. It is influenced by flooding attack (denial of service (DoS)) and fraudulent resource consumption (FRC) attack. Accountability implies capability of identifying a party responsible for the specific event. Several threats to cloud accountability are SLA violation, inaccurate billing, etc. Since cloud user's data and business logic reside among distributed cloud servers, there are potential risks that confidential and personal information will be disclosed to public or business competitors. Privacy preservation is one of the critical attributes of privacy, which prevents disclosure of private data. Confidentiality is indispensable, whereas integrity guarantees

data, or computation is corruption-free, which somehow preserves privacy.

The following challenges are faced by cloud providers to build secure and trustworthy cloud system.

- *Outsourcing*: A cloud user outsourced its content to the cloud provider's data center and lost the direct physical control over the data. The computing and data storage are done at the cloud provider's end, causing the main reason for cloud insecurity. Outsourcing may also incur privacy violations.
- *Multi-tenancy*: Since multiple customers share the cloud resources, it gives rise to co-residency issues. Cross-VM or co-resident VM attack exploits the multi-tenancy nature.
- *Massive data and intense computation*: Traditional security methods may not suffice for massive data storage and computation.

Various Attacks on VM

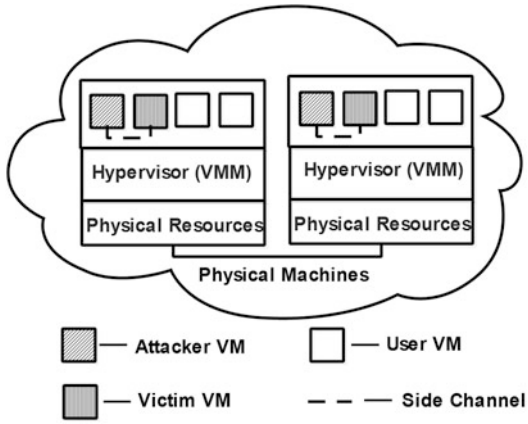
VMs are the principal computing unit in the cloud computing environment that facilitate efficient utilization of hardware resources and reduce maintenance overhead of computing resources. VMs provide the additional layer of hardware abstraction and isolation but require quality methods to prevent attackers to access information from other VMs and the host. Vulnerabilities in VMs pose an immediate threat to the privacy and security of the cloud services. The shared and distributed nature of cloud resources makes it difficult to develop a security standard that ensures data security and privacy. Vulnerabilities in the cloud are the security loopholes, which allow unauthorized access to the network and infrastructure resources. Vulnerabilities in virtualization or hypervisor will enable an attacker to perform a cross-VM side-channel attack, and DoS attack provides access to legitimate user's VM. Solution directives for VM attacks include alert message for any breach of VMs isolation, integrity and security assurance of VM images, vulnerability-free virtual machine manager (VMM) or hypervisor, etc. (Modi et al. 2013). Different classes of attacks (Hyde 2009) that affect VMs are:

- *Co-resident attack*: Attacker adversary VM to communicate with other colocated VMs on the same host, thus violating isolation feature of VMs.
- *Attack on hypervisor*: The attack on a hypervisor allows the attacker to access VM, host operating system, and hardware.
- *DoS attack*: DoS attacks are endeavors by an unapproved user to degrade or deny resources to a genuine user. DoS attack consumes resources from all VMs on the host.

Co-resident Attack

Colocated VMs are logically separated from each other. Users are migrating to the cloud that was exposed to an additional threat caused by neighbors due to the sharing of resources. This exposure raises questions about neighbors' trustworthiness and integrity as a malicious user can build side channel to circumvent logical isolation and extract sensitive information from co-resident VMs (Han et al. 2017). Like traditional web services, VMs are exposed to various security threats, and co-resident attack is one of them. Other names of co-resident attack are co-residence, co-residency, and colocation attack. Figure 2 shows co-resident VM attack in cloud. It shows that attacker VM and target VM are placed on the same physical machine, where attacker extracts target VM information through the side channel.

In co-resident attack, malicious users extract information from colocated virtual machines on the same physical machine by building side channels. The various side-channel attack includes cache attack, timing attack, power monitoring attack, software-initiated fault attack, etc. In cache attack, attackers take action based on cache access of the victim in a virtualized environment like a cloud, whereas timing attack is based on computation time of various activities (e.g., the comparison between attacker's password with



Co-resident Attack in Cloud Computing: An Overview, Fig. 2 Co-resident attack

victim's unknown one). The power monitoring attack makes use of power consumption variation by the hardware during computation. Software-initiated fault attack is a rare one, where off-limits memory is changed by accessing adjacent memory too often.

The co-resident attack is a two-step process. First, the attacker set target VMs and colocates their VMs with these VMs on the corresponding physical servers. The second process builds the side channels to extract valuable information from the victim. The following steps are performed by an attacker for co-resident attack (Han et al. 2016):

1. An attacker first targets some VMs.
2. Attacker starts some VMs either from a single account or multiple accounts.
3. Check whether new VM is colocated with target VM on the same physical machine or not.
4. Repeat steps 2–3 until co-residency is achieved.
5. If co-residency is achieved, non-colocated VMs are turned off. Now, the attacker gains access and steals information from the target VM.

Defense Methods

Cloud security providers mainly aim to prevent the attacker from achieving co-residence. Co-residence may lead to cross-VM information leakage. The attacker analyzes the cloud infrastructure and identifies the residency of target VM. After several attempts, an attacker may succeed to instantiate new VMs, co-resident with target VM. Now, the attacker can leak all kinds of information which are known as the side-channel attack. This is because both target and attacker VM share the same resource. Various methods used to defend co-resident attack in cloud are listed below:

- *Elimination of side channels*: One of the solutions to handle co-resident attack is the elimination of side channels. But, this method needs modifications to the current cloud platform.
- *VM-allocation policy*: Another answer is VM allocation policy, which can influence the probability of colocation. The bigger the probability of non-co-residency, the better the policy is.
- *Increasing difficulty level of co-residence verification*: Usually the IP address of a VM Dom0 is a privileged VM responsible for management of all VMs on a host. For co-resident VMs, Dom0 IP addresses must be same. So, the disclosure of Dom0 IP address must be restricted.
- *Detection of co-resident attacks trademark*: Any attack can be identified by unusual behavior of a cloud system. Various indicators of co-resident attack include abnormalities in the CPU and RAM behavior, cache miss, system calls, etc.
- *Periodic VM migration*: Periodic migration of VMs helps to counter co-resident attack. Like every coin has two sides, this method comes with additional drawbacks like extra power consumption, performance degradation, SLA violation, etc.
- *Encrypted VM images*: Encrypted VM images can guarantee the security of a VM, which can be accessible by only an authorized user who knows the security keys of permitted access.

Case Study

Multiple VMs of various organizations run on Amazon's EC2 infrastructures with virtual boundaries, and each VM can operate within a single physical server. VMs located on the same physical server have IP addresses approaching each other. An attacker can set up lots of his own virtual machines to get a glance at their IP addresses and figure out VMs that share the same resources as a meditated target. Once the adversary VM is located along with target VM on the same server, an attacker can deduce sensitive information about the victim, and this action can cause resource usage fluctuations.

Conclusion

This chapter manifests various security and privacy issues of cloud running big data applications. Virtualization technology and VM are the two most essential components of a cloud. Our aim in this paper is to give an overview of various security issues in the cloud (specific to VM) and its solutions. Here, we present details about co-resident VM attack that includes steps performed by an attacker, various defense methods, and the explanation with an example.

References

Bhargava B, Çho Y, Kim A (2010) Research in cloud security and privacy. https://www.cs.purdue.edu/homes/bb/cloud/Security-Privacy.ppt

Cloud-Security-Alliance (2012) Top ten big data security and privacy challenges. https://downloads.cloudsecurityalliance.org/initiatives/bdwg/Big_Data_Top_Ten_v1.pdf

Han Y, Alpcan T, Chan J, Leckie C, Rubinstein BIP (2016) A game theoretical approach to defend against co-resident attacks in cloud computing: preventing co-residence using semi-supervised learning. IEEE Trans Inf Forensics Secur 11(3):556–570

Han Y, Chan J, Alpcan T, Leckie C (2017) Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. IEEE Trans Dependable Secure Comput 14(1):95–108

Hyde D (2009) A survey on the security of virtual machines. www.cse.wustl.edu/jain/cse571-09/ftp/vmsec/index.html

Inukollu VN, Arsi S, Ravuri SR (2014) Security issues associated with big data in cloud computing. Int J Netw Secur Appl 6(3):45

Mishra SK, Putha D, Rodrigues JJ, Sahoo B, Dutkiewicz E (2018a, in press) Sustainable service allocation using metaheuristic technique in fog server for industrial applications. IEEE Trans Ind Inf

Mishra SK, Puthal D, Sahoo B, Jena SK, Obaidat MS (2018b) An adaptive task allocation technique for green cloud computing. J Supercomput 74(1):370–385

Modi C, Patel D, Borisaniya B, Patel A, Rajarajan M (2013) A survey on security issues and solutions at different layers of cloud computing. J Supercomput 63(2):561–592

Puthal D, Sahoo B, Mishra S, Swain S (2015) Cloud computing features, issues, and challenges: a big picture. In: Computational Intelligence and Networks (CINE), 2015 International Conference on, IEEE, pp 116–123

Sahoo S, Nawaz S, Mishra SK, Sahoo B (2015) Execution of real time task on cloud environment. In: India Conference (INDICON), 2015 Annual IEEE, IEEE, pp 1–5

Sahoo S, Sahoo B, Turuk AK, Mishra SK (2016) Real time task execution in cloud using mapreduce framework. In: Resource management and efficiency in cloud computing environments, IGI Global, p 190

Xiao Z, Xiao Y (2013) Security and privacy in cloud computing. IEEE Commun Surv Tutorials 15(2):843–859

Zhang Y, Juels A, Oprea A, Reiter MK (2011) Homealone: co-residency detection in the cloud via side-channel analysis. In: 2011 IEEE Symposium on Security and Privacy, pp 313–328

CRUD Benchmarks

Steffen Friedrich and Norbert Ritter
Department of Informatics, University of Hamburg, Hamburg, Germany

Synonyms

NoSQL benchmarks

Definitions

A CRUD benchmark is a generic experimental framework for characterizing and comparing the performance of database systems by executing

workloads using only a simple CRUD (create, read, update, and delete) interface.

Since each database system provides at least this minimal subset of operations, CRUD benchmarks allow a general performance comparison of a variety of non-relational *NoSQL database systems*.

Overview

For many decades, relational database management systems (RDBMSs) have been the first choice for business applications. A high degree of standardization, especially the standard query language SQL, made it possible to develop domain-specific benchmarks for RDBMSs.

The DebitCredit benchmark by Jim Gray laid down the foundation for standardized database benchmarks (Anon et al. 1985). It introduced most of the key ideas that were later taken up by the Transaction Processing Performance Council (TPC) for the specification of the TPC-A benchmark and its famous successor, TPC-C (2010). The TPC was founded in 1988 as an industry-consensus body to fulfill the need for benchmark certification. Previously, many RDBMS vendors had conducted benchmarks based on their own interpretation of DebitCredit. Since its foundation, the TPC has developed numerous benchmarks for various kinds of online transactional processing (OLTP) and also online analytical processing (OLAP) workloads.

However, the amount of useful data in some applications has become so vast that it cannot be stored or processed by traditional relational database solutions (Hu et al. 2014). At the beginning of the twenty-first century, this led to the development of non-relational database systems able to cope with such Big Data. These systems are subsumed under the term *NoSQL database systems*. They offer horizontal scalability and high availability by sacrificing querying capabilities, the strict ACID guarantees, and the transactional support as known from RDBMSs.

Since most TPC benchmark specifications assume ACID compliant transactions, they cannot be applied to mostly non-transactional NoSQL databases. Defining similar domain-specific standards for those systems is even more complicated due to the sheer heterogeneity in their data models and APIs.

Therefore, many NoSQL vendors and open source communities implemented their own database-specific performance measurement tools, just as the RDBMS vendors did before TPC. These include, for example, the Cassandra-stress tool, Mongoperf for MongoDB, the Redis-benchmark utility, or Basho_bench for Riak. In the end, the results of these tools are often difficult to understand, cannot be extended to other scenarios, and do not allow comparisons between different databases.

Eventually, the *Yahoo! Cloud Serving Benchmark (YCSB)* was proposed (Cooper et al. 2010). YCSB is limited to a simple CRUD interface and has become the de facto standard for NoSQL database performance comparison, which is why the term *NoSQL Benchmark* is often used interchangeably for CRUD benchmark. Due to the fact that YCSB is open source and extensible, the developer community has added support for almost every NoSQL database system since the release in 2010.

Scientific Fundamentals

The Benchmark Handbook for Database and Transaction Systems edited by Jim Gray (1993) can be regarded as the first reference work of database benchmarking. It identifies four requirements that a domain-specific benchmark must meet:

1. **Relevance** means that the benchmarks should be based on a realistic workload such that the results reflect something important to their readers.
2. **Simplicity** describes that the workload should be understandable.
3. **Portability** means that it should be easy to run the benchmark against a large number of systems.

4. **Scalability** says that it should be possible to scale the benchmark up to larger systems as performance and architectures evolve.

These requirements have been refined over time to reflect the current developments of database systems (Huppler 2009; Folkerts et al. 2013). The book *Cloud Service Benchmarking* by Bermbach et al. (2017) is an excellent modern reference work in which these design objectives are also discussed in detail with regard to cloud computing and Big Data requirements. Regarding the mentioned four requirements, CRUD benchmarks significantly weaken the relevance aspect in order to achieve the greatest possible portability, simplicity, and scalability. They do not model a real domain-specific application workload like the TPC standard specifications but a simpler one in which the choice of CRUD operations is selected according to a given probability distribution of real web application workloads (*workload generation*). The typical *metrics* measured during the benchmark run are throughput (operations per second) and latency (response time in milliseconds). CRUD benchmarks also define hardly any restrictions on the *system under test* in favor of portability. The actual data values consist of simple synthetically generated random character strings (*data generation*). Overall, this makes them less relevant for concrete applications but allows to measure the performance of highly scalable NoSQL systems in the first place.

Hence, CRUD benchmarks are also often used and extended in research to measure additional performance aspects of distributed NoSQL database systems. An overview of these research efforts can be found in Friedrich et al. (2014), and the book by Bermbach et al. (2017) can also be recommended as a reference for the current state of measurement methods for nonfunctional properties such as scalability, elasticity, and consistency of NoSQL data management solutions.

It should be noted that the basics of scalability and elasticity benchmarking were already determined by the Wisconsin benchmark, an early performance evaluation framework for RDBMS (DeWitt 1993). It was by no means as popular as DebitCredit and the TPC benchmarks for comparative studies by RDBMS vendors, but it was widely used in research for measuring the performance of parallel database system prototypes designed for shared nothing hardware architectures. As these database prototypes should already meet horizontal scalability and elasticity requirements, the benchmark defined the key metrics scale-up and speedup to measure them.

Current research work is further developing domain-specific benchmarks for certain classes of NoSQL database systems such as document stores that go beyond pure CRUD benchmarks. This allows to compare the performance of more sophisticated query capabilities in the context of a particular application. Reniers et al. (2017) give an up-to-date overview of these domain-specific NoSQL benchmarks and CRUD benchmarks.

Key Applications

In addition to the four requirements that a benchmark must meet, Gray (1993) also describes four motivations for database benchmarking which can be extended to NoSQL database performance evaluation:

1. **Different proprietary software and hardware systems:** The classical competitive situation between two relational database vendors running their solution on their own dedicated specialized hardware can, nowadays, for example, be transferred to proprietary NoSQL database as a service solutions (DBaaS) as provided by cloud service providers.
2. **Different software products on the same hardware:** This is the most common form in which NoSQL database system vendors compare their systems with others. The different databases are set up in a cluster of the same hardware or cloud infrastructure, and their performance is usually measured with a selection of different workloads.
3. **The same software product on different hardware systems:** The database workload serves as a reference application for comparing different underlying hardware platforms.

Nowadays, this can mean that users benchmark their NoSQL database system on different infrastructure as a service (IaaS) solutions to select the most performant or cost-effective cloud provider.

4. **Different releases of a software product on the same hardware:** The benchmarking of newer versions of a database system is an integral part of the modern agile software development process. This is one of the major reasons why NoSQL developers often implement database-specific performance measurement tools.

Comparative studies are primarily promoted by database vendors themselves for marketing and advertising purposes. Since there is no instance like TPC for the verification of benchmark results in the NoSQL area, these results should always be looked at critically, because the workload parameters, the hardware, and the presentation of the results are selected in such a way that the own product performs particularly well. This kind of *benchmarketing* already existed for RDBMSs in the early 1980s.

In addition to comparative studies, one of the main motivations for benchmarking is systems research, i.e., understanding the performance behavior of certain system designs or understanding quality trade-offs like the consistency/latency trade-off of different data replication techniques.

Future Directions

The research in the field of benchmarking nonfunctional properties of distributed NoSQL database systems is still in an early stage. For example, a number of methods for measuring the consistency of replicated databases have already been proposed. Apart from further studies on scalability and elasticity, there is only little work on the topic of availability benchmarking of NoSQL database systems, especially not under different failure conditions. Therefore, further research in this area is still needed to develop standard methodologies for measuring these requirements.

Furthermore, there is a variety of possible benchmark designs between simple CRUD benchmarks and application-oriented benchmarks for one specific NoSQL data model that still need to be explored. One can imagine domain-specific workloads that are still general enough to allow specific implementations for all the different data models.

Cross-References

- NOSQL Database Systems
- System Under Test
- YCSB

References

Anon, Bitton D, Brown M, Catell R, Ceri S, Chou T, DeWitt D, Gawlick D, Garcia-Molina H, Good B, Gray J, Homan P, Jolls B, Lukes T, Lazowska E, Nauman J, Pong M, Spector A, Trieber K, Sammer H, Serlin O, Stonebraker M, Reuter A, Weinberger P (1985) A measure of transaction processing power. Datamation 31(7):112–118

Bermbach D, Wittern E, Tai S (2017) Cloud service benchmarking: measuring quality of cloud services from a client perspective. Springer, Cham

Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on cloud computing, SoCC'10. ACM, New York, pp 143–154

DeWitt DJ (1993) The wisconsin benchmark: past, present, and future. In: Gray J (ed) The benchmark handbook. Morgan Kaufmann, San Mateo

Folkerts E, Alexandrov A, Sachs K, Iosup A, Markl V, Tosun C (2013) Benchmarking in the cloud: what it should, can, and cannot be. Springer, Berlin/Heidelberg, pp 173–188

Friedrich S, Wingerath W, Gessert F, Ritter N (2014) NoSQL OLTP benchmarking: a survey. In: 44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data – Komplexität meistern, 22–26 Sept 2014 in Stuttgart, pp 693–704

Gray J (ed) (1993) The benchmark handbook for database and transaction systems, 2nd edn. Morgan Kaufmann, San Mateo

Hu H, Wen Y, Chua TS, Li X (2014) Toward scalable systems for big data analytics: a technology tutorial. IEEE Access 2:652–687

Huppler K (2009) The art of building a good benchmark. Springer, Berlin/Heidelberg, pp 18–30

Reniers V, Van Landuyt D, Rafique A, Joosen W (2017) On the state of NoSQL benchmarks. In: Proceedings of the 8th ACM/SPEC on international conference on performance engineering companion, ICPE'17 companion. ACM, New York, pp 107–112

TPC-C (2010) Benchmark specification. http://www.tpc.org/tpcc

Cryptocurrency

▶ Blockchain Transaction Processing