Chapter 18
Current Issues: Streaming Data and Cloud Computing

In this chapter we discuss two topics that are of growing importance in database management. The topics are data stream management (Section 18.1) and cloud data management (Section 18.2). Both of these topics have been topics of considerable interest in the community in recent years. They are still evolving, but there is a possibility that they may have considerable commercial impact. Our objective in this chapter is to give a snapshot of where the field is with respect to these systems at this point, and discuss potential research directions.

18.1 Data Stream Management

The database systems that we have discussed until now consist of a set of unordered objects that are relatively static, with insertions, updates and deletions occurring less frequently than queries. They are sometimes called *snapshot databases* since they show a snapshot of the values of data objects at a given point in time. Queries over these systems are executed when posed and the answer reflects the current state of the database. In these systems, typically, the data are persistent and queries are transient.

However, the past few years have witnessed an emergence of applications that do not fit this data model and querying paradigm. These applications include, among others, sensor networks, network traffic analysis, financial tickers, on-line auctions, and applications that analyze transaction logs (such as web usage logs and telephone call records). In these applications, data are generated in real time, taking the form of an unbounded sequence (stream) of values. These are referred to as the *data stream* applications. In this section, we discuss systems that support these applications; these systems are referred to as *data stream management systems* (DSMS).

A fundamental assumption of the data stream model is that new data are generated continually and in fixed order, although the arrival rates may vary across applications from millions of items per second (e.g., Internet traffic monitoring) down to several items per hour (e.g., temperature and humidity readings from a weather monitoring station). The ordering of streaming data may be implicit (by arrival time at the processing site) or explicit (by generation time, as indicated by a *timestamp* appended to each data item by the source). As a result of these assumptions, DSMSs face the following novel requirements.

1. Much of the computation performed by a DSMS is push-based, or data-driven. Newly arrived stream items are continually (or periodically) pushed into the system for processing. On the other hand, a DBMS employs a mostly pull-based, or query-driven computation model, where processing is initiated when a query is posed.
2. As a consequence of the above, DSMS queries are *persistent* (also referred to as continuous, long-running, or standing queries) in that they are issued once, but remain active in the system for a possibly long period of time. This means that a stream of updated results must be produced as time goes on. In contrast, a DBMS deals with one-time queries (issued once and then "forgotten"), whose results are computed over the current state of the database.
3. The system conditions may not be stable during the *lifetime* of a persistent query. For example, the stream arrival rates may fluctuate and the query workload may change.
4. A data stream is assumed to have unbounded, or at least unknown, length. From the system's point of view, it is infeasible to store an entire stream in a DSMS. From the user's point of view, recently arrived data are likely to be more accurate or useful.
5. New data models, query semantics and query languages are needed for DSMSs in order to reflect the facts that streams are ordered and queries are persistent.

The applications that generate streams of data also have similarities in the type of operations that they perform. We list below a set of fundamental continuous query operations over streaming data.

- **Selection:** All streaming applications require support for complex filtering.
- **Nested aggregation:** Complex aggregates, including nested aggregates (e.g., comparing a minimum with a running average) are needed to compute trends in the data.
- **Multiplexing and demultiplexing:** Physical streams may need to be decomposed into a series of logical streams and conversely, logical streams may need to be fused into one physical stream (similar to group-by and union, respectively).
- **Frequent item queries:** These are also known as *top-k* or *threshold* queries, depending on the cut-off condition.
- **Stream mining:** Operations such as pattern matching, similarity searching, and forecasting are needed for on-line mining of streaming data.
- **Joins:** Support should be included for multi-stream joins and joins of streams with static meta-data.

- **Windowed queries:** All of the above query types may be constrained to return results inside a window (e.g., the last 24 hours or the last one hundred packets).

Proposed data stream systems resemble the abstract architecture shown in Figure 18.1. An input monitor regulates the input rates, perhaps by dropping items if the system is unable to keep up. Data are typically stored in three partitions: temporary working storage (e.g., for window queries that will be discussed shortly), summary storage for stream synopses, and static storage for meta-data (e.g., physical location of each source). Long-running queries are registered in the query repository and placed into groups for shared processing, though one-time queries over the current state of the stream may also be posed. The query processor communicates with the input monitor and may re-optimize the query plans in response to changing input rates. Results are streamed to the users or temporarily buffered. Users may then refine their queries based on the latest results.

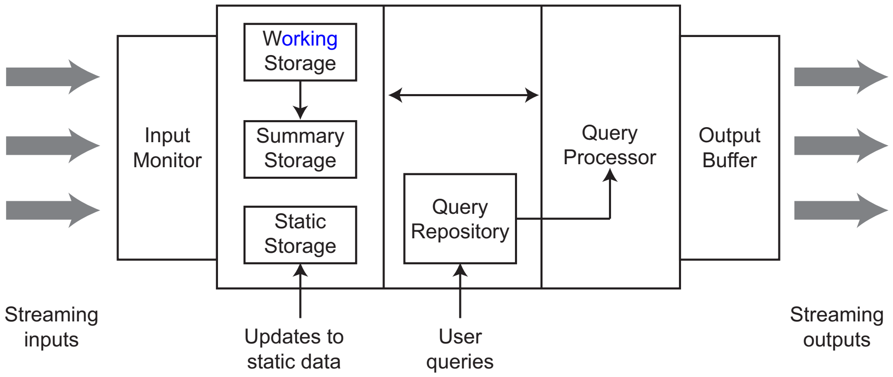

Fig. 18.1 Abstract reference architecture for a data stream management system.

18.1.1 Stream Data Models

A data stream is an append-only sequence of timestamped items that arrive in some order [Guha and McGregor, 2006]. While this is the commonly accepted definition, there are more relaxed versions; for example, *revision tuples*, which are understood to replace previously reported (presumably erroneous) data [Ryvkina et al., 2006], may be considered so that the sequence is not append-only. In publish/subscribe systems, where data are produced by some sources and consumed by those who subscribe to those data feeds, a data stream may be thought of as a sequence of events that are being reported continually [Wu et al., 2006]. Since items may arrive in bursts, a stream may instead be modeled as a sequence of sets (or bags) of elements [Tucker et al., 2003], with each set storing elements that have arrived during the same unit of time (no order is specified among tuplesthat have arrived at the same time). In relation-based stream models (e.g., STREAM [Arasu et al., 2006]), individual items take the form of relational tuples such that all tuples arriving on the same stream have the same schema. In object-based models (e.g., COUGAR [Bonnet et al., 2001] and Tribeca [Sullivan and Heybey, 1998]), sources and item types may be instantiations of (hierarchical) data types with associated methods. Stream items may contain explicit source-assigned timestamps or implicit timestamps assigned by the DSMS upon arrival. In either case, the timestamp attribute may or may not be part of the stream schema, and therefore may or may not be visible to users. Stream items may arrive out of order (if explicit timestamps are used) and/or in pre-processed form. For instance, rather than propagating the header of each IP packet, one value (or several partially pre-aggregated values) may be produced to summarize the length of a connection between two IP addresses and the number of bytes transmitted. This gives rise to the following list of possible models [Gilbert et al., 2001]:

1. *Unordered cash register*: Individual items from various domains arrive in no particular order and without any pre-processing. This is the most general model.
2. *Ordered cash register*: Individual items from various domains are not pre-processed but arrive in some known order, e.g., timestamp order.
3. *Unordered aggregate*: Individual items from the same domain are pre-processed and only one item per domain arrives in no particular order, e.g., one packet per TCP connection.
4. *Ordered aggregate*: Individual items from the same domain are pre-processed and one item per domain arrives in some known order, e.g., one packet per TCP connection in increasing order of the connection end-times.

As discussed earlier, unbounded streams cannot be stored locally in a DSMS, and only a recent excerpt of a stream is usually of interest at any given time. In general, this may be accomplished using a *time-decay model* [Cohen and Kaplan, 2004; Cohen and Strauss, 2003; Douglis et al., 2004], also referred to as an *amnesic* [Palpanas et al., 2004] or *fading* [Aggarwal et al., 2004] model. Time-decay models discount each item in the stream by a scaling factor that is non-decreasing with time. Exponential and polynomial decay are two examples, as are window models where items within the window are given full consideration and items outside the window are ignored. Windows may be classified according the the following criteria.

1. *Direction of movement of the endpoints:* Two fixed endpoints define a *fixed window*, two sliding endpoints (either forward or backward, replacing old items as new items arrive) define a *window!sliding*, and one fixed endpoint and one moving endpoint (forward or backward) define a *window!landmark*. There are a total of nine possibilities as each of the two endpoints could be fixed, moving forward, or moving backward.

2. *Definition of window size:* Logical, or *time-based* windows are defined in terms of a time interval, whereas physical, (also known as *count-based* or *tuple-based*) windows are defined in terms of the number of tuples. Moreover, *partitioned windows* may be defined by splitting a sliding window into groups and defining a separate count-based window on each group [Arasu et al., 2006]. The most general type is a *predicate window*, in which an arbitrary predicate specifies the contents of the window; e.g., all the packets from TCP connections that are currently open [Ghanem et al., 2006]. A predicate window is analogous to a materialized view.
3. *Windows within windows:* In the *elastic window model*, the maximum window size is given, but queries may need to run over any smaller window within the boundaries of the maximum window [Zhu and Shasha, 2003]. In the *n-of-N window model*, the maximum window size is N tuples or time units, but any smaller window of size n and with one endpoint in common with the larger window is also of interest [Lin et al., 2004].
4. *Window update interval:* Eager updating advances the window upon arrival of each new tuple or expiration of an old tuple, but batch processing (lazy updating) induces a *jumping window*. Note that a count-based window may be updated periodically and a time-based window may be updated after some number of new tuples have arrived; these are referred to as *mixed jumping windows* [Ma et al., 2005]. If the update interval is larger than the window size, then the result is a series of non-overlapping *tumbling windows* [Abadi et al., 2003].

As a consequence of the unbounded nature of data streams, DSMS data models may include some notion of change or drift in the underlying distribution that is assumed to generate the attribute values of stream items [Kifer et al., 2004; Dasu et al., 2006; Zhu and Ravishankar, 2004]. We will come back to this issue when we discuss data stream mining in Section 18.1.8. Additionally, it has been observed that in many practical scenarios, the stream arrival rates and distributions of values tend to be bursty or skewed [Kleinberg, 2002; Korn et al., 2006; Leland et al., 1994; Paxson and Floyd, 1995; Zhu and Shasha, 2003].

18.1.2 Stream Query Languages

Earlier we indicated that stream queries are usually persistent. So, one issue to discuss is what the semantics of these queries are, i.e., how do they generate answers. Persistent queries may be monotonic or non-monotonic. A *monotonic query* is one whose results can be updated incrementally. In other words, if $Q(t)$ is the answer to a query at time t, given two executions of the query at t_i and t_j, $Q(t_i) \subseteq Q(t_j)$ for all $t_j > t_i$. For monotonic queries, one can define the following:

$$Q(t) = \bigcup_{t_i=1}^{t} (Q(t_i) - Q(t_{i-1})) \cup Q(0)$$

That is, it is sufficient to re-evaluate the query over newly arrived items and append qualifying tuples to the result [Arasu et al., 2006]. Consequently, the answer of a monotonic persistent query is a continuous, append-only stream of results. Optionally, the output may be updated periodically by appending a batch of new results. It has been proven that a query is monotonic if and only if it is *non-blocking*, which means that it does not need to wait until the end-of-output marker before producing results [Law et al., 2004].

Non-monotonic queries may produce results that cease to be valid as new data are added and existing data changed (or deleted). Consequently, they may need to be re-computed from scratch during every re-evaluation, giving rise to the following semantics:

$$Q(t) = \bigcup_{t_i=0}^{t} Q(t_i)$$

Let us now consider classes of languages that have been proposed for DSMSs. Three querying paradigms can be identified: declarative, object-based, and procedural. *Declarative languages* have SQL-like syntax, but stream-specific semantics, as described above. Similarly, *object-based languages* resemble SQL in syntax, but employ DSMS-specific constructs and semantics, and may include support for streaming abstract data types (ADTs) and associated methods. Finally, *procedural languages* construct queries by defining data flow through various operators.

18.1.2.1 Declarative Languages

The languages in this class include CQL [Arasu et al., 2006; Arasu and Widom, 2004a], GSQL [Cranor et al., 2003], and StreaQuel [Chandrasekaran et al., 2003]. We discuss each of them briefly.

The Continuous Query Language (CQL) is used in the STREAM DSMS and includes three types of operators: relation-to-relation (corresponding to standard relational algebraic operators), stream-to-relation (*sliding windows*), and relation-to-stream. Conceptually, unbounded streams are converted to relations by way of sliding windows, the query is computed over the current state of the sliding windows as if it were a traditional SQL query, and the output is converted back to a stream. There are three relation-to-stream operators—`Istream`, `Dstream`, and `Rstream`—which specify the nature of the output. The `Istream` operator returns a stream of all those tuples which exist in a relation at the current time, but did not exist at the current time minus one. Thus, `Istream` suggests incremental evaluation of monotonic queries. `Dstream` returns a stream of tuples that existed in the given relation in the previous time unit, but not at the current time. Conceptually, `Dstream` is analogous to generating negative tuples for non-monotonic queries. Finally, the `Rstream` operator streams the contents of the entire output relation at the current time and corresponds to generating the complete answer of a non-monotonic query. The `Rstream` operator may also be used in periodic query evaluation to produce an output stream consisting of a sequence of relations, each corresponding to the answer at a different point in time.

Example 18.1. Computing a join of two time-based windows of size one minute each, can be performed by the following query:

```
SELECT Rstream(*)
FROM   S1 [RANGE 1 min], S2 [RANGE 1 min]
WHERE  S1.a = S2.a
```

The `RANGE` keyword following the name of the input stream specifies a time-based sliding window on that stream, whereas the `ROWS` keyword may be used to define count-based sliding windows. ♦

GSQL is used in Gigascope, a stream database for network monitoring and analysis. The input and output of each operator is a stream for reasons of composability. Each stream is required to have an ordering attribute, such as timestamp or packet sequence number. GSQL includes a subset of the operators found in SQL, namely selection, aggregation with group-by, and join of two streams, whose predicate must include ordering attributes that form a join window. The *stream merge* operator, not found in standard SQL, is included and works as an order-preserving union of ordered streams. This operator is useful in network traffic analysis, where flows from multiple links need to be merged for analysis. Only landmark windows are supported directly, but sliding windows may be simulated via user-defined functions.

StreaQuel is used in the TelegraphCQ system and is noteworthy for its windowing capabilities. Each query, expressed in SQL syntax and constructed from SQL's set of relational operators, is followed by a for-loop construct with a variable `t` that iterates over time. The loop contains a `WindowIs` statement that specifies the type and size of the window. Let `S` be a stream and let `ST` be the start time of a query. To specify a sliding window over `S` with size five that should run for fifty time units, the following for-loop may be appended to the query.

```
for(t=ST; t<ST+50; t++)
  WindowIs(S, t-4, t)
```

Changing to a landmark window can be done by replacing `t-4` with some constant in the `WindowIs` statement. Changing the for-loop increment condition to `t=t+5` would cause the query to re-execute every five time units. The output of a StreaQuel query consists of a time sequence of sets, each set corresponding to the answer set of the query at that time.

18.1.2.2 Object-Based Languages

One approach to object-oriented stream modeling is to classify stream contents according to a type hierarchy. This method is used in the Tribeca network monitoring system, which implements Internet protocol layers as hierarchical data types [Sullivan and Heybey, 1998]. The query language used in Tribeca has SQL-like syntax, but accepts a single stream as input, and returns one or more output streams. Supported operators are limited to projection, selection, aggregation over the entire input stream or over a sliding window, multiplex and demultiplex (corresponding to union and group-by respectively, except that different sets of operators may be applied on each of the demultiplexed sub-streams), as well as a join of the input stream with a fixed window.

Another object-based possibility is to model the sources as ADTs, as in the COUGAR system for managing sensor data [Bonnet et al., 2001]. Each type of sensor is modeled by an ADT, whose interface consists of the supported signal processing methods. The proposed query language has SQL-like syntax and also includes a `$every()` clause that indicates the query re-execution frequency. However, few details on the language are available in the published literature and therefore it is not included in Figure 18.2.

Example 18.2. A simple query that runs every sixty seconds and returns temperature readings from all sensors on the third floor of a building may be specified as follows:

```
SELECT R.s.getTemperature()
FROM   R
WHERE  R.floor = 3 AND $every(60)
```

♦

18.1.2.3 Procedural Languages

An alternative to declarative query languages is to let the user specify how the data should flow through the system. In the Aurora DSMS [Abadi et al., 2003], users construct query plans via a graphical interface by arranging boxes, corresponding to query operators, and joining them with directed arcs to specify data flow, though the system may later re-arrange, add, or remove operators in the optimization phase. SQuAl is the boxes-and-arrows query language used in Aurora, which accepts streams as inputs and returns streams as output (however, static data sets may be incorporated into query plans via *connection points* [Abadi et al., 2003]). There are a total of seven operators in the SQuAl algebra, four of them order-sensitive. The three order-insensitive operators are projection, union, and `map`, the last applying an arbitrary function to each of the tuples in the stream or a window thereof. The other four operators require an order specification, which includes the ordered field and a slack parameter. The latter defines the maximum disorder in the stream, e.g., a slack of 2 means that each tuple in the stream is either in sorted order, or at most two positions or two time units away from being in sorted order. The four order-sensitive operators are buffered sort (which takes an almost-sorted stream and the slack parameter, and outputs the stream in sorted order), windowed aggregates (in which the user can specify how often to advance the window and re-evaluate the aggregate), binary band join (which joins tuples whose timestamps are at most t units apart), and resample (which generates missing stream values by interpolation, e.g., given tuples with timestamps 1 and 3, a new tuple with timestamp 2 can be generated with an attribute value that is an average of the other two tuples' values. Other resampling functions are also possible, e.g., the maximum, minimum, or weighted average of the two neighbouring data values.

18.1.2.4 Summary of DSMS Query Languages

A summary of the proposed DSMS query languages is provided in Figure 18.2 with respect to the allowed inputs and outputs (streams and/or relations), novel operators, supported window types (fixed, landmark or sliding), and supported query re-execution frequency (continuous and/or periodic). With the exception of SQuAl, the surface syntax of DSMS query languages is similar to SQL, but their semantics are considerably different. CQL allows the widest range of semantics with its relation-to-stream operators; note that CQL uses the semantics of SQL during its relation-to-relation phase and incorporates streaming semantics in the stream-to-relation and relation-to-stream components. On the other hand, GSQL, SQuAL, and Tribeca only allow streaming output, whereas StreaQuel continually (or periodically) outputs the entire answer set. In terms of expressive power, CQL closely mirrors SQL as CQL's core set of operators is identical to that of SQL. Additionally, StreaQuel can express a wider range of windows than CQL. GSQL, SQuAl, and Tribeca, which operate in the stream-in-stream-out mode, may be thought of as restrictions of SQL as they focus on incremental, non-blocking computation. In particular, GSQL and Tribeca are application-specific (network monitoring) and have been designed for very fast implementation [Cranor et al., 2003]. However, although SQuAl and GSQL are stream-in/stream-out languages, and, as a result, may have lost some expressive power as compared to SQL, they may regain this power via user-defined functions. Moreover, SQuAl is noteworthy for its attention to issues related to real-time processing such as buffering, out-of-order arrivals and timeouts.

Language/ system	Allowed inputs	Allowed outputs	Novel operators	Supported windows	Execution frequency
CQL/ STREAM	streams and relations	streams and relations	relation-to-stream, stream-to-relation	sliding	continuous or periodic
GSQL/ Gigascope	streams	streams	order-preserving union	landmark	periodic
SQuAl/ Aurora	streams and relations	streams	resample, `map`, buffered sort	fixed, landmark, sliding	continuous or periodic
StreaQuel/ TelegraphCQ	streams and relations	sequences of relations	`WindowIs`	fixed, landmark, sliding	continuous or periodic
Tribeca	single stream	streams	multiplex, demultiplex	fixed, landmark, sliding	continuous

Fig. 18.2 Summary of proposed data stream languages

18.1.3 Streaming Operators and their Implementation

While the streaming languages discussed above may resemble standard SQL, their implementation, processing, and optimization present novel challenges. In this section, we highlight the differences between streaming operators and traditional relational operators, including non-blocking behavior, approximations, and sliding windows. Note that simple relational operators such as projection and selection (that do not keep state information) may be used in streaming queries without any modifications.

Some relational operators are blocking. For instance, prior to returning the next tuple, the Nested Loops Join (NLJ) may potentially scan the entire inner relation and compare each tuple therein with the current outer tuple. Some operators have non-blocking counterparts, such as joins [Haas and Hellerstein, 1999a; Urhan and Franklin, 2000; Viglas et al., 2003; Wilschut and Apers, 1991] and simple aggregates [Hellerstein et al., 1997; Wang et al., 2003c]. For example, a pipelined symmetric hash join [Wilschut and Apers, 1991] builds hash tables on-the-fly for each of the participating relations. Hash tables are stored in main memory and when a tuple from one of the relations arrives, it is inserted into its table and the other tables are probed for matches. It is also possible to incrementally output the average of all the items seen so far by maintaining the cumulative sum and item count. When a new item arrives, the item count is incremented, the new item's value is added to the sum, and an updated average is computed by dividing the sum by the count. There remains the issue of memory constraints if an operator requires too much working memory, so a windowing scheme may be needed to bound the memory requirements. Hashing has also been used in developing join execution strategies over DHT-based P2P systems [Palma et al., 2009].

Another way to unblock query operators is to exploit constraints over the input streams. Schema-level constraints include synchronization among timestamps in multiple streams, clustering (duplicates arrive contiguously), and ordering [Babu et al., 2004b]. If two streams have nearly synchronized timestamps, an equi-join on the timestamp can be performed in limited memory: a *scrambling bound* B may be set such that if a tuple with timestamp τ arrives, then no tuple with timestamp greater than $\tau - B$ may arrive later [Motwani et al., 2003].

Constraints at the data level may take the form of control packets inserted into a stream, called *punctuations* [Tucker et al., 2003]. Punctuations are constraints (encoded as data items) that specify conditions for all future items. For instance, a punctuation may arrive asserting that all the items henceforth shall have the A attribute value larger than 10. This punctuation could be used to partially unblock a group-by query on A since all the groups where $A \leq 10$ are guaranteed not to change for the remainder of the stream's lifetime, or until another punctuation arrives and specifies otherwise. Punctuations may also be used to synchronize multiple streams in that a source may send a punctuation asserting that it will not produce any tuples with timestamp smaller than τ [Arasu et al., 2006].

As discussed above, unblocking a query operator may be accomplished by re-implementing it in an incremental form, restricting it to operate over a window (more on this shortly), and exploiting stream constraints. However, there may be cases where an incremental version of an operator does not exist or is inefficient to evaluate, where even a sliding window is too large to fit in main memory, or where no suitable stream constraints are present. In these cases, compact stream summaries may be stored and approximate queries may be posed over the summaries. This implies a trade-off between accuracy and the amount of memory used to store the summaries. An additional restriction is that the processing time per item should be kept small, especially if the inputs arrive at a fast rate.

Counting methods, used mainly to compute quantiles and frequent item sets, typically store frequency counts of selected item types (perhaps chosen by sampling) along with error bounds on their true frequencies. Hashing may also be used to summarize a stream, especially when searching for frequent items—each item type may be hashed to n buckets by n distinct hash functions and may be considered a potentially frequent flow if all of its hash buckets are large. Sampling is a well known data reduction technique and may be used to compute various queries to within a known error bound. However, some queries (e.g., finding the maximum element in a stream) may not be reliably computed by sampling.

Sketches were initially proposed by Alon et al. [1996] and have since then been used in various approximate algorithms. Let $f(i)$ be the number of occurrences of value i in a stream. A sketch of a data stream is created by taking the inner product of f with a vector of random values chosen from some distribution with a known expectation. Moreover, wavelet transforms (that reduce the underlying signal to a small set of coefficients) have been proposed to approximate aggregates over infinite streams.

We end this section with a discussion of window operators. Sliding window operators process two types of events: arrivals of new tuples and expirations of old tuples; the orthogonal problem of determining when tuples expire will be discussed in the next section. The actions taken upon arrival and expiration vary across operators [Hammad et al., 2003b; Vossough and Getta, 2002]. A new tuple may generate new results (e.g., join) or remove previously generated results (e.g., negation). Furthermore, an expired tuple may cause a removal of one or more tuples from the result (e.g., aggregation) or an addition of new tuples to the result (e.g., duplicate elimination and negation). Moreover, operators that must explicitly react to expired tuples (by producing new results or invalidating existing results) perform state purging eagerly (e.g., duplicate elimination, aggregation, and negation), whereas others may do so eagerly or lazily (e.g., join).

In a sliding window join, newly arrived tuples on one of the inputs probe the state of the other input, as in a join of unbounded streams. Additionally, expired tuples are removed from the state [Golab and Özsu, 2003b; Hammad et al., 2003a, 2005; Kang et al., 2003; Wang et al., 2004]. Expiration can be done periodically (lazily), so long as old tuples can be identified and skipped during processing.

Aggregation over a sliding window updates its result when new tuples arrive and when old tuples expire. In many cases, the entire window needs to be stored in order to account for expired tuples, although selected tuples may sometimes be removed early if their expiration is guaranteed not to influence the result. For example, when computing `MAX`, tuples with value v need not be stored if there is another tuple in the window with value greater than v and a younger timestamp. Additionally, in order to enable incremental computation, the aggregation operator stores the current answer (for distributive and algebraic aggregates) or frequency counters of the distinct values present in the window (for holistic aggregates). For instance, computing `COUNT` requires storing the current count, incrementing it when a new tuple arrives, and decrementing it when a tuple expires. In this case, in contrast to the join operator, expirations must be dealt with immediately so that an up-to-date aggregate value can be returned right away.

Duplicate elimination over a sliding window may also produce new output when an input tuple expires. This occurs if a tuple with value v was produced on the output stream and later expires from its window, yet there are other tuples with value v still present in the window [Hammad et al., 2003b]. Alternatively, as is the case in the STREAM system, duplicate elimination may produce a single result tuple with a particular value v and retain it on the output stream so long as there is at least one tuple with value v present in the window. In both cases, expirations must be handled eagerly so that the correct result is maintained at all times.

Finally, negation of two sliding windows, $W_1 - W_2$, may produce *negative tuples* (e.g., arrival of a W_2-tuple with value v causes the deletion of a previously reported result with value v), but may also produce new results upon expiration of tuples from W_2 (e.g., if a tuple with value v expires from W_2, then a W_1-tuple with value v may need to be appended to the output stream [Hammad et al., 2003b]). There are methods for implementing duplicate-preserving negation, but those are beyond our scope in this chapter.

18.1.4 Query Processing

Let us now discuss the issues related to processing queries in DSMSs. The overall process is similar to relational systems: declarative queries are translated into execution plans that map logical operators specified in the query into physical implementations. For now, let us assume that the inputs and operator state fit in main memory; we will discuss disk-based processing later.

18.1.4.1 Queuing and Scheduling

DBMS operators are pull-based, whereas DSMS operators consume data pushed into the plan by the sources.

Queues allow sources to push data into the query plan and operators to retrieve data as needed [Abadi et al., 2003; Adamic and Huberman, 2000; Arasu et al., 2006; Madden and Franklin, 2002; Madden et al., 2002a]. A simple scheduling strategy allocates a time slice to each operator, during which the operator extracts tuples from its input queue(s), processes them in timestamp order, and deposits output tuples into the next operator's input queue. The time slice may be fixed or dynamically calculated based upon the size of an operator's input queue and/or processing speed. A possible improvement could be to schedule one or more tuples to be processed by multiple operators at once. In general, there are several possibly conflicting criteria involved in choosing a scheduling strategy, among them queue sizes in the presence of bursty stream arrival patterns [Babcock et al., 2004], average or maximum latency of output tuples [Carney et al., 2003; Jiang and Chakravarthy, 2004; Ou et al., 2005], and average or maximum delay in reporting the answer relative to the arrival of new data [Sharaf et al., 2005].

18.1.4.2 Determining When Tuples Expire

In addition to dequeuing and processing new tuples, sliding window operators must remove old tuples from their state buffers and possibly update their answers, as discussed in Section 18.1.3. Expiration from an individual time-based window is simple: a tuple expires if its timestamp falls out of the range of the window. That is, when a new tuple with timestamp ts arrives, it receives another timestamp, call it exp, that denotes its expiration time as ts plus the window length. In effect, every tuple in the window may be associated with a lifetime interval of length equal to the window size [Krämer and Seeger, 2005]. Now, if this tuple joins with a tuple from another window, whose insertion and expiration timestamps are ts' and exp', respectively, then the expiration timestamp of the result tuple is set to $\min(exp, exp')$. That is, a composite result tuple expires if at least one of its constituent tuples expires from its windows. This means that various join results may have different lifetime lengths and furthermore, the lifetime of a join result may have a lifetime that is shorter than the window size [Cammert et al., 2006]. Moreover, as discussed above, the negation operator may force some result tuples to expire earlier than their exp timestamps by generating negative tuples. Finally, if a stream is not bounded by a sliding window, then the expiration time of each tuple is infinity [Krämer and Seeger, 2005].

In a count-based window, the number of tuples remains constant over time. Therefore, expiration can be implemented by overwriting the oldest tuple with a newly arrived tuple. However, if an operator stores state corresponding to the output of a count-based window join, then the number of tuples in the state may change, depending upon the join attribute values of new tuples. In this case, expirations must be signaled explicitly using negative tuples.

18.1.4.3 Continuous Query Processing over Sliding Windows

There are two techniques for sliding window query processing and state maintenance: the negative tuple approach and the direct approach. In the negative tuple approach [Arasu et al., 2006; Hammad et al., 2003b, 2004], each window referenced in the query is assigned an operator that explicitly generates a negative tuple for every expiration, in addition to pushing newly arrived tuples into the query plan. Thus, each window must be materialized so that the appropriate negative tuples are produced. This approach generalizes the purpose of negative tuples, which are now used to signal all expirations explicitly, rather than only being produced by the negation operator if a result tuple expires because it no longer satisfies the negation condition. Negative tuples propagate through the query plan and are processed by operators in a similar way as regular tuples, but they also cause operators to remove corresponding "real" tuples from their state. The negative tuple approach can be implemented efficiently using hash tables as operator state so that expired tuples can be looked up quickly in response to negative tuples. Conceptually, this is similar to a DBMS indexing a table or materialized view on the primary key in order to speed up insertions and deletions. However, the downside is that twice as many tuples must be processed by the query because every tuple eventually expires from its window and generates a corresponding negative tuple. Furthermore, additional operators must be present in the plan to generate negative tuples as the window slides forward.

Direct approach [Hammad et al., 2003b, 2004] handles negation-free queries over time-based windows. These queries have the property that the expiration times of base tuples and intermediate results can be determined via their *exp* timestamps, as explained in Section 18.1.4.2. Hence, operators can access their state directly and find expired tuples without the need for negative tuples. The direct approach does not incur the overhead of negative tuples and does not have to store the base windows referenced in the query. However, it may be slower than the negative tuple approach for queries over multiple windows [Hammad et al., 2003b]. This is because straightforward implementations of state buffers may require a sequential scan during insertions or deletions. For example, if the state buffer is sorted by tuple arrival time, then insertions are simple, but deletions require a sequential scan of the buffer. On the other hand, sorting the buffer by expiration time simplifies deletions, but insertions may require a sequential scan to ensure that the new tuple is ordered correctly, unless the insertion order is the same as the expiration order.

18.1.4.4 Periodic Query Processing Over Sliding Windows

Query Processing over Windows Stored in Memory.

For reasons of efficiency (reduced expiration and query processing costs) and user preference (users may find it easier to deal with periodic output rather than a continuous output stream [Arasu and Widom, 2004b; Chandrasekaran and Franklin, 2003]), sliding windows may be advanced and queries re-evaluated periodically with a specified frequency [Abadi et al., 2003; Chandrasekaran et al., 2003; Golab et al., 2004; Liu et al., 1999]. As illustrated in Figure 18.3, a periodically-sliding window can be modeled as a circular array of *sub-windows*, each spanning an equal time interval for time-based windows (e.g., a ten-minute window that slides every minute) or an equal number of tuples for tuple-based windows (e.g., a 100-tuple window that slides every ten tuples).

Rather than storing the entire window and re-computing an aggregate after every new tuple arrives or an old tuple expires, a synopsis can be stored that pre-aggregates

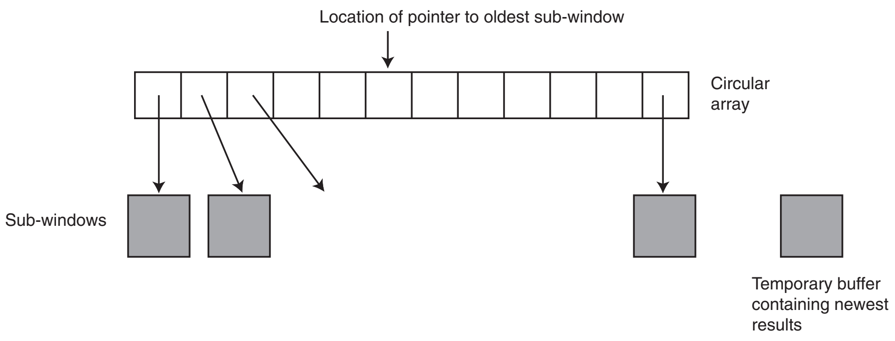
Location of pointer to oldest sub-window
Circular array
Sub-windows
Temporary buffer containing newest results

Fig. 18.3 Sliding window implemented as a circular array of pointers to sub-windows

each sub-window and reports updated answers whenever the window slides forward by one sub-window. Thus a "window update" occurs when the oldest sub-window is replaced with newly arrived data (accumulated in a buffer), thereby sliding the window forward by one sub-window. Depending on the type of operator one deals with, it would be necessary to use different types of synopsis (e.g., a *running synopsis* [Arasu and Widom, 2004b] for subtractable aggregates [Cohen, 2006] such as `SUM` and `COUNT` or an *interval synopsis* for distributive aggregates that are not subtractable, such as `MIN` and `MAX`). An aggregate f is subtractable if, for two multi-sets X and Y such that $X \supseteq Y$, $f(X-Y) = f(X) - f(Y)$.Details are beyond our scope in this chapter.

A disadvantage of periodic query evaluation is that results may be stale. One way to stream new results after each new item arrives is to bound the error caused by delayed expiration of tuples in the oldest sub-window. It has been shown [Datar et al., 2002] that restricting the sizes of the sub-windows (in terms of the number of tuples) to powers of two and imposing a limit on the number of sub-windows of each size yields a space-optimal algorithm (called *exponential histogram*, or EH) that approximates simple aggregates to within ε using logarithmic space (with respect to the sliding window size). Variations of the EH algorithm have been used to approximately compute the sum [Datar et al., 2002; Gibbons and Tirthapura, 2002], variance and k-medians clustering [Babcock et al., 2003], windowed histograms [Qiao et al., 2003], and order statistics [Lin et al., 2004; Xu et al., 2004]. Extensions of the EH algorithm to time-based windows have also been proposed [Cohen and Strauss, 2003].

18.1.4.5 Query Processing over Windows Stored on Disk.

In traditional database applications that use secondary storage, performance may be improved if appropriate indices are built. Consider maintaining an index over a periodically-sliding window stored on disk, e.g., in a data warehousing scenario where new data arrive periodically and decision support queries are executed (offline) over the latest portion of the data. In order to reduce the index maintenance costs, it is desirable to avoid bringing the entire window into memory during every update. This can be done by partitioning the data so as to localize updates (i.e., insertions of newly arrived data and deletion of tuples that have expired from the window) to a small number of disk pages. For example, if an index over a sliding window is partitioned chronologically [Folkert et al., 2005; Shivakumar and García-Molina, 1997], then only the youngest partition incurs insertions, while only the oldest partition needs to be checked for expirations (the remaining partitions "in the middle" are not accessed).The disadvantage of chronological clustering is that records with the same search key may be scattered across a very large number of disk pages, causing index probes to incur prohibitively many disk I/Os.

One way to reduce index access costs is to store a reduced (summarized) version of the data that fits on fewer disk pages [Chandrasekaran and Franklin, 2004], but this does not necessarily improve index update times. In order to balance the access and update times, a *wave index* has been proposed that chronologically divides a sliding window into *n* equal partitions, each of which is separately indexed and clustered by search key for efficient data retrieval [Shivakumar and García-Molina, 1997]. The window can be partitioned either by insertion time or by expiration time; these are equivalent from the perspective of wave indexes.

18.1.5 DSMS Query Optimization

It is usually the case that a query may be executed in a number of different ways. A DBMS query optimizer is responsible for enumerating (some or all of) the possible query execution strategies and choosing an efficient one using a cost model and/or a set of transformation rules. A DSMS query optimizer has the same responsibility, but it must use an appropriate cost model and rewrite rules. Additionally, DSMS query optimization involves adaptivity, load shedding, and resource sharing among similar queries running in parallel, as summarized below.

18.1.5.1 Cost Metrics and Statistics

Traditional DBMSs use selectivity information and available indices to choose efficient query plans (e.g., those which require the fewest disk accesses). However, this cost metric does not apply to (possibly approximate) persistent queries, where processing cost per-unit-time is more appropriate [Kang et al., 2003]. Alternatively, if the stream arrival rates and output rates of query operators are known, then it may be possible to optimize for the highest output rate or to find a plan that takes the least time to output a given number of tuples [Tao et al., 2005; Urhan and Franklin, 2001; Viglas and Naughton, 2002]. Finally, quality-of-service metrics such as response time may also be used in DSMS query optimization [Abadi et al., 2003; Berthold et al., 2005; Schmidt et al., 2004, 2005].

18.1.5.2 Query Rewriting and Adaptive Query Optimization

Some of the DSMS query languages discussed in Section 18.1.2 introduce rewritings for new operators, e.g., selections and time-based sliding windows commute, but not selections and count-based windows [Arasu et al., 2006]. Other rewritings are similar to those used in relational databases, e.g., re-ordering a sequence of binary joins in order to minimize a particular cost metric. There has been some work in join ordering for data streams in the context of the rate-based model [Viglas and Naughton, 2002; Viglas et al., 2003]. Furthermore, adaptive re-ordering of pipelined stream filters [Babu et al., 2004a] and adaptive materialization of intermediate join results [Babu et al., 2005] have been investigated.

The notion of adaptivity is important in query rewriting; operators may need to be re-ordered on-the-fly in response to changes in system conditions. In particular, the cost of a query plan may change for three reasons: change in the processing time of an operator, change in the selectivity of a predicate, and change in the arrival rate of a stream [Adamic and Huberman, 2000]. Initial efforts on adaptive query plans include mid-query re-optimization [Kabra and DeWitt, 1998] and query scrambling, where the objective was to pre-empt any operators that become blocked and schedule other operators instead [Amsaleg et al., 1996b; Urhan et al., 1998b]. To further increase adaptivity, instead of maintaining a rigid tree-structured query plan, the Eddy approach [Adamic and Huberman, 2000] performs scheduling of each tuple separately by routing it through the operators that make up the query plan. In effect, the query plan is dynamically re-ordered to match current system conditions. This is accomplished by tuple routing policies that attempt to discover which operators are fast and selective, and those operators are scheduled first. A recent extension adds queue length as the third factor for tuple routing strategies in the presence of multiple distributed Eddies [Tian and DeWitt, 2003a]. There is, however, an important trade-off between the resulting adaptivity and the overhead required to route each tuple separately. More details on adaptive query processing may be found in [Babu and Bizarro, 2005; Babu and Widom, 2004; Gounaris et al., 2002a].

Adaptivity involves on-line reordering of a query plan and may therefore require that the internal state stored by some operators be migrated over to the new query plan consisting of a different arrangement of operators [Deshpande and Hellerstein, 2004; Zhu et al., 2004]. We do not discuss this issue further in this chapter.

18.1.6 Load Shedding and Approximation

The stream arrival rates may be so high that not all tuples can be processed, regardless of the (static or run-time) optimization techniques used. In this case, two types of load shedding may be applied—random or semantic—with the latter making use of stream properties or quality-of-service parameters to drop tuples believed to be less significant than others [Tatbul et al., 2003]. For an example of semantic load shedding, consider performing an approximate sliding window join with the objective of attaining the maximum result size. The idea is that tuples that are about to expire or tuples that are not expected to produce many join results should be dropped (in case of memory limitations [Das et al., 2005; Li et al., 2006; Xie et al., 2005]), or inserted into the join state but ignored during the probing step (in case of CPU limitations [Ayad et al., 2006; Gedik et al., 2005; Han et al., 2006]). Note that other objectives are possible, such as obtaining a random sample of the join result [Srivastava and Widom, 2004].

In general, it is desirable to shed load in such a way as to minimize the drop in accuracy. This problem becomes more difficult when multiple queries with many operators are involved, as it must be decided where in the query plan the tuples should be dropped. Clearly, dropping tuples early in the plan is effective because all of the subsequent operators enjoy reduced load. However, this strategy may adversely affect the accuracy of many queries if parts of the plan are shared. On the other hand, load shedding later in the plan, after the shared sub-plans have been evaluated and the only remaining operators are specific to individual queries, may have little or no effect in reducing the overall system load.

One issue that arises in the context of load shedding and query plan generation is whether an optimal plan chosen without load shedding is still optimal if load shedding is used. It has been shown that this is indeed the case for sliding window aggregates, but not for queries involving sliding window joins [Ayad and Naughton, 2004].

Note that instead of dropping tuples during periods of high load, it is also possible to put them aside (e.g., spill to disk) and process them when the load has subsided [Liu et al., 2006; Reiss and Hellerstein, 2005]. Finally, note that in the case of periodic re-execution of persistent queries, increasing the re-execution interval may be thought of as a form of load shedding [Babcock et al., 2002; Cammert et al., 2006; Wu et al., 2005].

18.1.7 Multi-Query Optimization

As seen in Section 18.1.4.4, memory usage may be reduced by sharing internal data structures that store operator state [Denny and Franklin, 2005; Dobra et al., 2004; Zhang et al., 2005]. Additionally, in the context of complex queries containing stateful operators such as joins, computation may be shared by building a common query plan [Chen et al., 2000]. For example, queries belonging to the same group may share a plan, which produces the union of the results needed by the individual queries. A final selection is then applied to the shared result set and new answers are routed to the appropriate queries. An interesting trade-off appears between doing similar work multiple times and doing too much unnecessary work; techniques that balance this trade-off are presented in [Chen et al., 2002; Krishnamurthy et al., 2004; Wang et al., 2006]. For example, suppose that the workload includes several queries referencing a join of the same windows, but having a different selection predicate. If a shared query plan performs the join first and then routes the output to appropriate queries, then too much work is being done because some of the joined tuples may not satisfy any selection predicate (unnecessary tuples are being generated). On the other hand, if each query performs its selection first and then joins the surviving tuples, then the join operator cannot be shared and the same tuples will be probed many times.

For selection queries, a possible multi-query optimization is to index the query predicates and store auxiliary information in each tuple that identifies which queries it satisfies [Chandrasekaran and Franklin, 2003; Demers et al., 2006; Hanson et al., 1999; Krishnamurthy et al., 2006; Lim et al., 2006; Madden et al., 2002a; Wu et al., 2004]. When a new tuple arrives for processing, its attribute values are extracted and matched against the query index to see which queries are satisfied by this tuple. Data and queries may be thought of as duals, in some cases reducing query processing to a multi-way join of the query predicate index and the data tables [Chandrasekaran and Franklin, 2003; Lim et al., 2006].

18.1.8 Stream Mining

In addition to querying as discussed in the previous sections, mining of stream data has been studied for a number of applications. Data mining involves the use of data analysis tools to discover previously unknown relationships and patterns in large data sets. The characteristics of data streams discussed above impose new challenges in performing mining tasks; many of the well-known techniques cannot be used. The major issues are the following:

- **Unbounded data set.** Traditional data mining algorithms are based on the assumption that they can access the full data set. However, this is not possible in data streams, where only a portion of the old data is available and much of the old data are discarded. Hence, data mining techniques that require multiple scan over the entire data set cannot be used.
- **"Messy" data.** Data are never entirely clean, but in traditional data mining applications, they can be cleaned before the application is run. In many stream applications, due to the high arrival rates of data streams, this is not always possible. Given that in many cases the data that are read from sensors and other sources of stream data are already quite noisy, the problem is even more serious.
- **Real-time processing.** Data mining over traditional data is typically a batch process. Although there are obvious efficiency concerns in analyzing these data, they are not as severe as those on data streams. Since data arrival is continuous and potentially at high rate, the mining algorithms have to have real-time performance.
- **Data evolution.** As noted earlier traditional data sets can be assumed to be static, i.e., the data is a sample from a static distribution. However, this is not true for many real-world data streams, since they are generated over long

periods of time during which the underlying phenomena can change resulting in significant changes in the distribution of the data values. This means that some mining results that were previously generated may no longer be valid. Therefore, a data stream mining technique must have the ability to detect changes in the stream, and to automatically modify its mining strategy for different distributions.

In the remainder, we will summarize some stream mining techniques. We divide the discussion into two groups: general processing techniques, and specific data mining tasks and their algorithms [Gaber et al., 2005]. Data processing techniques are general approaches to process the stream data before specific tasks can be applied. These consist of the following:

Sampling. As discussed earlier, data stream sampling is the process of choosing a suitable representative subset from the stream of interest. In addition to the major use of stream sampling to reduce the potentially infinite size of the stream to a bounded set of samples, it can be utilized to clean "messy" data and to preserve representative sets for the historical distributions. However, since some data elements of the stream are not looked at, in general, it is impossible to guarantee that the results produced by the mining application using the samples will be identical to the results returned on the complete stream up to the most recent time. Therefore, one of the most critical tasks for stream sampling techniques is to provide guarantees about how much the results obtained using the samples differ from the non-sampling based results.

Load shedding. The arrival speed of elements in data streams are usually unstable, and many data stream sources are prone to dramatic spikes in load. Therefore, stream mining applications must cope with the effects of system overload. Maximizing the mining benefits under resource constraints is a challenging task. Load shedding techniques as discussed earlier are helpful.

Synopsis maintenance. Synopsis maintenance processes create synopses or "sketches" for summarizing the streams and were introduced earlier in this chapter. A synopsis does not represent all characteristics of a stream, but rather some "key features" that might be useful for tuning the stream mining processes and further analyzing the streams. It is especially useful for stream mining applications that are expecting various streams as input, or an input stream with frequent distribution changes. When the stream changes, some re-computation, either from scratch or incrementally, has to be done. An efficient synopsis maintenance process can generate summary of the stream shortly after the change, and the stream mining application can re-adjust its settings or switch to another mining technique based on these precious information.

Change detection. When the distribution of the stream changes, previous mining results may no longer be valid under the new distribution, and the mining technique must be adjusted to maintain good performance for the new distribution. Hence, it is critical for the distribution changes in a stream to be detected in real-time so that the stream mining application can react promptly.

There are basically two different tracks oftechniques for detecting changes. One track is to look at the natureof the dataset and determine if that set has evolved [Kifer et al., 2004; Aggarwal, 2003, 2005], and the othertrack is to detect if an existing data model is no longer suitablefor recent data, which implies the concept drifting [Hulten et al., 2001; Wang et al., 2003a; Fan, 2004; Gama et al., 2005]belong to the second track.

Now we take a look at some of the popular stream mining tasks and how they can be accomplished in this environment. We focus on clustering, classification, frequency counting and association rule mining, and time series analysis.

Clustering. Clustering groups together data with similar behavior. It can be thought of as partitioning or segmenting elements into groups (clusters) that may or may not be disjoint. In many cases, the answer to a clustering problem is not unique, i.e., many answers can be found, and interpreting the practical meaning of each cluster may be difficult.

Aggarwal et al. [2003] have proposed a framework for clustering data streams that uses an online component to store summarized information about the streams, and an offline component that performs clustering on the summarized data. This framework has been extended in HPStream in a way that can find projected clusters for high dimensional data streams [Aggarwal et al., 2004] .

The existing clustering algorithms can be categorized into decision tree based ones (e.g., [Domingos and Hulten, 2000; Gama et al., 2005; Hulten et al., 2001; Tao and Özsu, 2009]) and k-mean (or k-median) based approaches (e.g., [Babcock et al., 2002; Charikar et al., 1997, 2003; Guha et al., 2003; Ordonez, 2003]).

Classification. Classification maps data into predefined groups (classes). Its difference from clustering is that, in classification, the number of groups is predetermined and fixed. Similar to clustering, classification techniques can also adopt the decision tree model (e.g., [Ding et al., 2002; Ganti et al., 2002]). Two decision tree classifiers — Interval Classifier [Agrawal et al., 1992] and SPRINT [Shafer et al., 1996] — can mine databases that do not fit in main memory, and are thus are suitable for data streams. The VFDT [Domingos and Hulten, 2000] and CVFDT [Hulten et al., 2001] systems originally designed for stream clustering can also be adopted for classification tasks.

Frequency counting and association rule mining. The problem of frequency counting, and mining association rules (frequent itemsets) has long been recognized as an important issue. However, although mining frequent itemsets has been widely studied in data mining and a number of efficient algoirthms exist, extending these to data streams is challenging, especially for streams with non-static distributions [Jiang and Gruenwald, 2006].

Mining frequent itemsets is a continuous process that runs throughout a stream's life span. Since the total number of itemsets is exponential, making it impractical to keep count of each itemset in order to incrementally adjust the frequent itemsets as new data items arrive. Usually only the itemsets that are already known to be frequent are recorded and monitored, and counters of infrequent itemsets are discarded [Chakrabarti et al., 2002; Cormode and Muthukrishnan, 2003; Demaine et al., 2002; Halatchev and Gruenwald, 2005] . However, since data streams can change over time, an itemset that was once infrequent may become frequent if the distribution changes. Such (new) frequent itemsets are difficult to detect, since mining data streams is a one-pass procedure and history information is not retrievable.

Time series analysis. In general, a time series is a set of attribute values over a period of time. Usually a time series consists of only numeric values, either continuous or discrete. Consequently, it is possible to model data streams that contain only numeric values as time series. This allows one to use analysis techniques that have been developed on time series for some types of stream data. Mining tasks over time series can be briefly classified into two types: pattern detection and trend analysis. A typical mining task for pattern detection is the following: given a sample pattern or a base time series with a certain pattern, find all the time series that contain this pattern. The tasks for trend prediction are detecting trends in time series and predicting the upcoming trends.

18.2 Cloud Data Management

Cloud computing is the latest trend in distributed computing and has been the subject of much hype. The vision encompasses on demand, reliable services provided over the Internet (typically represented as a cloud) with easy access to virtually infinite computing, storage and networking resources. Through very simple web interfaces and at small incremental cost, users can outsource complex tasks, such as data storage, system administration, or application deployment, to very large data centers operated by cloud providers. Thus, the complexity of managing the software/hardware infrastructure gets shifted from the users' organization to the cloud provider.

Cloud computing is a natural evolution, and combination, of different computing models proposed for supporting applications over the web: service oriented architectures (SOA) for high-level communication of applications through web services, utility computing for packaging computing and storage resources as services, cluster and virtualization technologies to manage lots of computing and storage resources, autonomous computing to enable self-management of complex infrastructure, and grid computing to deal with distributed resources over the network. However, what makes cloud computing unique is its ability to provide various levels of functionality such as infrastructure, platform, and application as services that can be combined to best fit the users' requirements [Cusumano, 2010]. From a technical point of view, the grand challenge is to support in a cost-effective way, the very large scale of the infrastructure that has to manage lots of users and resources with high quality of service.

Cloud computing has been developed by web industry giants, such as Amazon, Google, Microsoft and Yahoo, to create a new, huge market. Virtually all computer industry players are interested in cloud computing. Cloud providers have developed new, proprietary technologies (e.g., Google File System), typically with specific, simple applications in mind. There are already open source implementations (e.g., Hadoop Distributed File System) with much contribution from the research community. As the need to support more complex applications increases, the interest of the research community is steadily growing. In particular, data management in cloud computing is becoming a major research direction which we think can capitalize on distributed and parallel database techniques.

The rest of this section is organized as follows. First, we give a general taxonomy of the different kinds of clouds, and a discussion of the advantages and potential disadvantages. Second, we give an overview of grid computing, with which cloud computing is sometimes confused, and point out the main differences. Third, we present the main cloud architectures and associated functions. Fourth, we present the current solutions for data management in the cloud, in particular, data storage, database management and parallel data processing. Finally, we discuss open issues in cloud data management.

18.2.1 Taxonomy of Clouds

In this section, we first give a definition of cloud computing, with the main categories of cloud services. Then, we discuss the main data-intensive applications that are suitable for the cloud and the main issues, in particular, security.

Agreeing on a precise definition of cloud computing is difficult as there are many different perspectives (business, market, technical, research, etc.). However, a good working definition is that a “cloud provides on demand resources and services over the Internet, usually at the scale and with the reliability of a data center” [Grossman and Gu, 2009]. This definition captures well the main objective (providing on-demand resources and services over the Internet) and the main requirements for supporting them (at the scale and with the reliability of a data center).

Since the resources are accessed through services, everything gets delivered as a service. Thus, as in the services industry, this enables cloud providers to propose a pay-as-you-go pricing model, whereby users only pay for the resources they consume. However, implementing a pricing model is complex as users should be charged based on the level of service actually delivered, e.g., in terms of service availability or performance. To govern the use of services by customers and support pricing, cloud providers use the concept of Service Level Agreement (SLA), which is critical in the services industry (e.g., in telecoms), but in a rather simple way. The SLA (between the cloud provider and any customer) typically specifies the responsabilities, guarantees and service commitment. For instance, the service commitment might state that the service uptime during a billing cycle (e.g., a month) should be at least 99%, and if the commitment is not met, the customer should get a service credit.

Cloud services can be divided in three broad categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS).

- **Infrastructure-as-a-Service (IaaS).** IaaS is the delivery of a computing infrastructure (i.e., computing, networking and storage resources) as a service. It enables customers to scale up (add more resources) or scale down (release resources) as needed (and only pay for the resources consumed). This important capability is called *elasticity* and is typically achieved through *server virtualization*, a technology that enables multiple applications to run on the same physical server as virtual machines, i.e., as if they would run on distinct physical servers. Customers can then requisition computing instances as virtual machines and add and attach storage as needed. An example of popular IaaS is Amazon web Services.
- **Software-as-a-Service (SaaS).** SaaS is the delivery of application software as a service. It generalizes the earlier Application Service Provider (ASP) model whereby the hosted application is fully owned, operated and maintained by the ASP. With SaaS, the cloud provider allows the customer to use hosted applications (as with ASP) but also provides tools to integrate other applications, from different vendors or even developed by the customer (using the cloud platform). Hosted applications can range from simple ones such as email and calendar to complex applications such as customer relationship management (CRM), data analysis or even social networks. An example of popular SaaS is Safesforce CRM system.
- **Platform-as-a-Service (PaaS).** PaaS is the delivery of a computing platform with development tools and APIs as a service. It enables developers to create and deploy custom applications directly on the cloud infrastructure, in virtual machines, and integrate them with applications provided as SaaS. An example of popular PaaS is Google Apps.

By using a combination of IaaS, SaaS and PaaS, customers could move all or part of their information technology (IT) services to the cloud, with the following main benefits:

- **Cost.** The cost for the customer can be greatly reduced since the IT infrastructure does not need to be owned and managed; billing is only based only on resource consumption. For the cloud provider, using a consolidated infrastructure and sharing costs for multiple customers reduces the cost of ownership and operation.
- **Ease of access and use.** The cloud hides the complexity of the IT infrastructure and makes location and distribution transparent. Thus, customers can have access to IT services anytime, and from anywhere with an Internet connection.
- **Quality of Service (QoS).** The operation of the IT infrastructure by a specialized provider that has extensive experience in running very large infrastructures (including its own infrastructure) increases QoS.
- **Elasticity.** The ability to scale resources out, up and down dynamically to accommodate changing conditions is a major advantage. In particular, it makes it easy for customers to deal with sudden increases in loads by simply creating more virtual machines.

However, not all corporate applications are good candidates for being “cloudified” [Abadi, 2009]. To simplify, we can classify corporate applications between the two main classes of data-intensive applications which we already discussed: OLTP and OLAP. Let us recall their main characteristics. OLTP deals with operational databases of average sizes (up to a few terabytes), that are write-intensive, and require complete ACID transactional properties, strong data protection and response time guarantees. On the other hand, OLAP deals with historical databases of very large sizes (up to petabytes), that are read-intensive, and thus can accept relaxed ACID properties. Furthermore, since OLAP data are typically extracted from operational OLTP databases, sensitive data can be simply hidden for analysis (e.g., using anonymization) so that data protection is not as crucial as in OLTP.

OLAP is more suitable than OLTP for cloud primarily because of two cloud characteristics (see the detailed discussion in [Abadi, 2009]): elasticity and security. To support elasticity in a cost-effective way, the best solution, which most cloud providers adopt, is a shared-nothing cluster architecture. Recall from Section 14.1 that shared-nothing provides high-scalability but requires careful data partitioning. Since OLAP databases are very large and mostly read-only, data partitioning and parallel query processing are effective. However, it is much harder to support OLTP on shared-nothing because of ACID guarantees, which require complex concurrency control. For these reasons and because OLTP databases are not so large, shared-disk is the preferred architecture for OLTP. The second reason that OLTP is not so suitable for cloud is that the corporate data get stored at an untrusted host (the provider site). Storing corporate data at an untrusted third-party, even with a carefully negotiated SLA with a reliable provider, creates resistance from some customers because of security issues. However, this resistance is much reduced for historical data, and with anonymized sensitive data.

There are currently two main solutions to address the security issue in clouds: internal cloud and virtual private cloud. The mainstream cloud approach is generally called *public cloud*, because the cloud is available to anyone on the Internet. An *internal cloud* (or *private cloud*) is the use of cloud technologies for managing a company's data center, but in a private network behind a firewall. This brings much tighter security and many of the advantages of cloud computing. However, the cost advantage tends to be much reduced because the infrastructure is not shared with other customers. Nonetheless, an attractive compromise is the *hybrid cloud* which connects the internal cloud (e.g., for OLTP) with one or more public clouds (e.g., for OLAP). As an alternative to internal clouds, cloud providers such as Amazon and Google have proposed *virtual private clouds* with the promise of a similar level of security as an internal cloud, but within a public cloud. A virtual private cloud provides a Virtual Private Network (VPN) with security services to the customers. Virtual private clouds can also be used to develop hybrid clouds, with tighter security integration with the internal cloud.

One earlier criticism of cloud computing is that customers get locked in proprietary clouds. It is true that most clouds are proprietary and there are no standards for cloud interoperability. But this is changing with open source cloud software such as Hadoop, an Apache project implementing Google's major cloud services such as Google File System and MapReduce, and Eucalyptus, an open source cloud software infrastructure, which are attracting much interest from research and industry.

18.2.2 Grid Computing

Like cloud computing, grid computing enables access to very large compute and storage resources over the web. It has been the subject of much research and development over the last decade. Cloud computing is somewhat more recent and there are similarities but also differences between the two computing models. In this section, we discuss the main aspects of grid computing and end with a comparison with cloud computing.

Grid computing has been initially developed for the scientific community as a generalization of cluster computing, typically to solve very large problems (that require a lot of computing power and/or access to large amounts of data) using many computers over the web. Grid computing has also gained some interest in enterprise information systems. For instance, IBM and Oracle (since Oracle 10g with g standing for grid) have been promoting grid computing with tools and services for both scientific and enterprise applications.

Grid computing enables the virtualization of distributed, heterogeneous resources using web services [Atkinson et al., 2005]. These resources can be data sources (files, databases, web sites, etc.), computing resources (multiprocessors, supercomputers, clusters) and application resources (scientific applications, information management services, etc.). Unlike the web, which is client-server oriented, the grid is demand-oriented: users send requests to the grid which allocates them to the most appropriate resources to handle them. A grid is also an organized, secured environment managed and controlled by administrators. An important unit of control in a grid is the Virtual Organization (VO), i.e., a group of individuals, organizations or companies that share the same resources, with common rules and access rights. A grid can have one or more VOs, and may have different size, duration and goal.

Compared with cluster computing, which only deals with parallelism, the grid is characterized with high heterogeneity, large-scale distribution and large-scale parallelism. Thus, it can offer advanced services on top of very large amounts of distributed data.

Depending on the contributed resources and the targeted applications, many different kinds of grids and architectures are possible. The earlier computational grids typically aggregate very powerful sites (supercomputers, clusters) to provide high-performance computing for scientific applications (e.g., physics, astronomy). Data grids aggregate heterogeneous data sources (like a distributed database) and provide additional services for data discovery, delivery and use to scientific applications. More recently, enterprise grids [Jiménez-Peris et al., 2007] have been proposed to aggregate information system resources, such as web servers, application servers and database servers, in the enterprise.

Figure 18.4 illustrates a typical grid scenario, inspired by the Grid5000 platform in France, with two computing sites (clusters 1 and 2) and one storage site (cluster 3) accessible to authorized users. Each site has one cluster with service nodes and either compute or storage nodes. Service nodes provide common services for users (access, resource reservation, deployment) and administrators (infrastructure services) and are available at each site, through the replication of directories and catalogs.

Compute nodes provide the main computing power while storage nodes provide storage capacity (i.e., lots of disks). The basic communication between grid sites (e.g., to deploy an application or a system image) is through web services (WS) calls (to be discussed shortly). But for distributing computation between compute nodes at two different sites, communication is typically through the standard Message Passing Interface (MPI).

A typical scenario for solving a large scientific problem P is the following. P is initially decomposed (by a scientist programmer User 1) into two subproblems P_1 and P_2, each being solved through a parallel program to be run at one computing site. If P_1 and P_2 are independent then there is no need for communication between the computing sites. If there are computing dependencies, e.g., P_2 consumes results of P_1, communication between P_1 and P_2 must be specified and implemented through MPI. The data produced by P_1 and P_2 could then be sent to the storage site, typically using WS calls. To run P on the grid, a user must first reserve the computing resources (e.g., a needed number of cluster nodes at site 1 and 2) and storage resources (at site 3), deploy the jobs corresponding to the programs, and then start their parallel executions at site 1 and 2, which will produce data and send them to site 3. The resource allocation and the scheduling of job executions at the clusters are done by the grid middleware in a way that guarantees fair access to the reserved resources. More complex scenarios can also involve the distributed execution of workflows. On the other hand, User 2 can simply reserve storage capacity and use it for saving her local data (using the store interface).

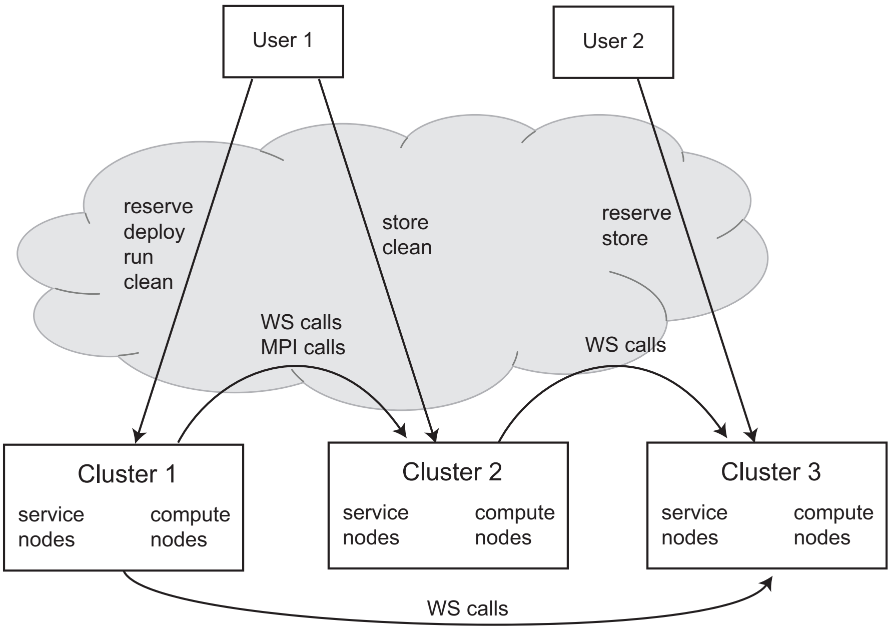

Fig. 18.4 A Grid Scenario

A common need of different kinds of grids is interoperability of heterogeneous resources. To address this need, the Globus Alliance, which represents the grid community, has defined the Open Grid Services Architecture (OGSA) as a standard SOA and a framework to create grid solutions using WS standards. OGSA provides three main layers to build grid applications: (1) resources layer, (2) web services layer, and (3) high-level grid services layer. The first layer provides an abstraction of the physical resources (servers, storage, network) that are managed by logical resources such as database systems, file systems, or workflow managers, all encapsulated by WS. The second layer extends WS, which are typically stateless, to deal with stateful grid services, i.e., those that can retain data between multiple invocations. This capability is useful for instance to access a resources state, e.g., the load of a server, through WS. Stateful grid services can be created and destroyed (using a grid service factory), and have an internal state which can be observed or even changed after notifications from other grid services. The third layer provides high-level grid-specific services such as resource provisioning, data management, security, workflow, and monitoring to ease the development and management of grid applications.

The adoption of WS in enterprise information systems has made OGSA appealing and several offerings for enterprise grids are based on the Globus platform (e.g., Oracle 11g). Web service standards are useful for grid data management: XML for data exchange, XMLSchema for schema description, Simple Object Access Protocol (SOAP) for remote procedure calls, UDDI for directory access, Web Service Definition Language (WSDL) for data source description, WS-Transaction for distributed transactions, Business Process Execution Language (BPEL) for workflow control, etc.

The main solutions for grid data management, in the context of computational grids, are file-based [Pacitti et al., 2007b]. A basic solution, used in Globus, is to combine global directory services to locate files and a secure file transfer protocol. Although simple, this solution does not provide distribution transparency as it requires the application to explicitly transfer files. Another solution is to use a distributed file system for the grid that can provide location-independent file access and transparent replication [Zhang and Honeyman, 2008].

Recent solutions have recognized the need for high-level data access and extended the distributed database architecture whereby clients send database requests to a grid multidatabase server that forwards them transparently to the appropriate database servers. These solutions rely on some form of global directory management, where directories can be distributed and replicated. In particular, users are able to use a high-level query language (SQL) to describe the desired data as with OGSA-DAI (OGSA Database Access and Integration), an OGSA standard for accessing and integrating distributed data [Antonioletti et al., 2005]. OGSA-DAI is a popular multidatabase system that provides uniform access to heterogeneous data sources (e.g., relational databases, XML databases or files) via WS within grids. Its architecture is similar to the mediator/wrapper architecture described in Chapters 1and 9 with the wrappers implemented by WS. The OGSA-DAI mediator includes a distributed query processor which automatically transforms a multidatabase query into a distributed QEP that specifies the WS calls to get the required data from each database wrapper.

We end this section with a discussion of the advantages and disadvantages of grid computing. The main advantages come from the distributed architecture when it uses clusters at each site, as it provides scalability, performance (through parallelism) and availability (through replication). It is also a cost-effective alternative to a huge supercomputer to solve larger, more complex problems in a shorter time. Another advantage is that existing resources are better used and shared with other organizations. The main disadvantages also come from the highly distributed architecture, which is complex for both administrators and developers. In particular, sharing resources across administrative domains is a political challenge for participating organizations as it is hard to assess their cost/benefits.

Compared with cloud computing, there are important differences in terms of objectives and architecture. Grid computing fosters collaboration among participating organizations to leverage existing resources whereas cloud computing provides a rather fixed (distributed) infrastructure to all kinds of users (and customers). Thus, SLA and pay-per-use are essential in cloud computing. The grid architecture is potentially much more distributed than the cloud architecture that typically consists of a few sites in different geographical regions, but each site being a very huge data center. Therefore, the scalability issue at a site (in terms of numbers of users or numbers of server nodes) is much harder in cloud computing. Finally, a major difference is that there are no standards such as OGSA for cloud interoperability.

18.2.3 Cloud architectures

Unlike in grid computing, there is no standard cloud architecture and there will probably never be one, since different cloud providers will provide different cloud services (IaaS, PaaS, SaaS) in different ways (public, private, virtual private, ...) depending on their business models. Thus, in this section, we discuss the main cloud architectures in order to identify the underlying technologies and functions. This is useful to be able to focus on data management (in the next section).

Figure 18.5 illustrates a typical cloud scenario, inspired by that of a popular IaaS/PaaS provider. This scenario is also useful for comparison with the typical grid scenario in Figure 18.4. We assume one cloud provider with two sites, each with the same capabilities and cluster architecture. Thus, any user can access any site to get the needed service as if there were only one site, so the cloud appears “centralized”. This is one major difference with grid as distribution can be completely hidden. However, distribution happens under the cover, e.g., to replicate data automatically from one site to the other in order to resist to site failure. Then, to solve the large scientific problem P, User 1 now does not need to decompose it into two subproblems, but she does need to provide a parallel version of P to be run at Site 1. This is done by creating a *virtual machine* (VM) (sometimes called *computing instance*) with executable application code and data, then starting as many VMs as needed for the parallel execution and finally terminating. User 1 is then charged only for the resources (VMs) consumed. The allocation of VMs to physical machines at Site 1 is done by the cloud middleware in a way that optimizes global resource consumption while satisfying the SLA. On the other hand, similar to the grid scenario, User 2 can also reserve storage capacity and use it for saving her local data.

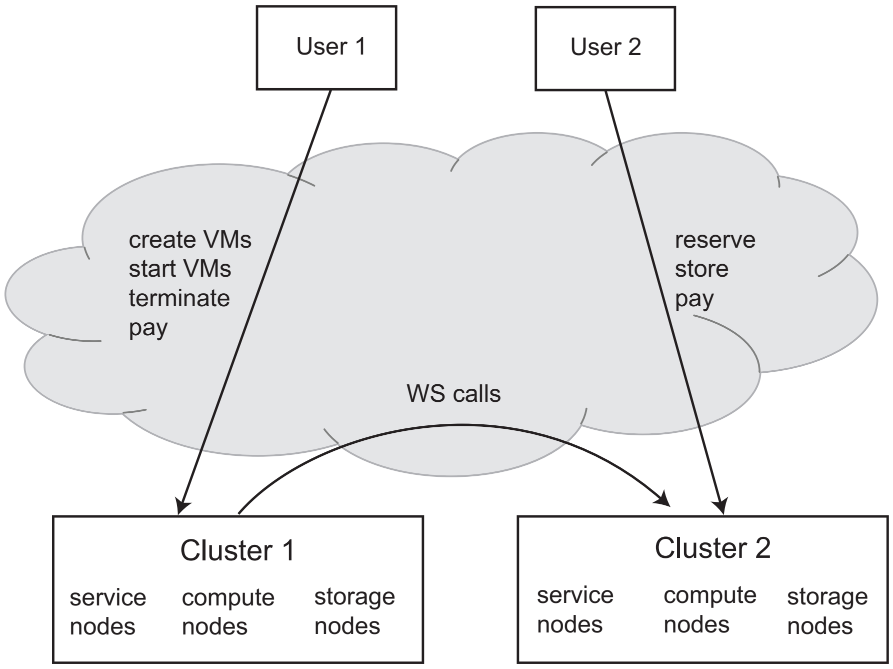

Fig. 18.5 A Cloud Scenario

We can distinguish the cloud architectures between infrastructure (IaaS) and software/platform (SaaS/PaaS). All architectures can be supported by a network of shared-nothing clusters. For IaaS, the preferred architectural model derives from the need to provide computing instances on demand. To support computing instances on demand, as in the scenario in Figure 18.5, the main solution is to rely on server virtualization, which enables VMs to be provisioned and decommissioned as needed. Server virtualization can be well supported by a shared-nothing cluster architecture. For SaaS/PaaS, many different architectural models can be used depending on the targeted services and applications. For instance, to support enterprise applications, a typical architecture is n-tier with web servers, application servers, database servers and storage servers, all organized in a cluster architecture. Server virtualization can also be used in such architecture. For data storage virtualization, SAN can be used to provide shared-disk access to service or compute nodes. As for grids, communication between applications and services is typically done through WS or message passing.

The main functions provided by clouds are similar to those found in grids: security, directory management, resource management (provisioning, allocation, monitoring) and data management (storage, file management, database management, data replication). In addition, clouds provide support for pricing, accounting and SLA management.

18.2.4 Data management in the cloud

For managing data, cloud providers could rely on relational DBMS technology, all of which have distributed and parallel versions. However, relational DBMSs have been lately criticized for their "one size fits all" approach [Stonebraker, 2010]. Although they have been able to integrate support for all kinds of data (e.g., multimedia objects, XML documents) and new functions, this has resulted in a loss of performance, simplicity and flexibility for applications with specific, tight performance requirements. Therefore, it has been argued that more specialized DBMS engines are needed. For instance, column-oriented DBMSs [Abadi et al., 2008], which store column data together rather than rows in traditional row-oriented relational DBMSs, have been shown to perform more than an order of magnitude better on OLAP workloads. Similarly, as discussed in Section 18.1, DSMSs are specifically architected to deal efficiently with data streams which traditional DBMS cannot even support.

The "one size does not fit all" argument generally applies to cloud data management as well. However, internal clouds or virtual private clouds for enterprise information systems, in particular for OLTP, may use traditional relational DBMS technology. On the other hand, for OLAP workloads and web-based applications on the cloud, relational DBMS provide both too much (e.g., ACID transactions, complex query language, lots of tuning parameters), and too little (e.g., specific optimizations for OLAP, flexible programming model, flexible schema, scalability) [Ramakrishnan, 2009]. Some important characteristics of cloud data have been considered for designing data management solutions. Cloud data can be very large (e.g., text-based or scientific applications), unstructured or semi-structured, and typically append-only (with rare updates). And cloud users and application developers may be in high numbers, but not DBMS experts. Therefore, current cloud data management solutions have traded consistency for scalability, simplicity and flexibility.

In this section, we illustrate cloud data management with representative solutions for distributed file management, distributed database management and parallel database programming.

18.2.4.1 Distributed File Management

The Google File System (GFS) [Ghemawat et al., 2003] is a popular distributed file system developed by Google for its internal use. It is used by many Google applications and systems, such as Bigtable and MapReduce, which we discuss next. There are also open source implementations of GFS, such as Hadoop Distributed File System (HDFS), a popular Java product.

Similar to other distributed file systems, GFS aims at providing performance, scalability, fault-tolerance and availability. However, the targeted systems, shared-nothing clusters, are challenging as they are made of many (e.g., thousands of) servers built from inexpensive hardware. Thus, the probability that any server fails at a given time is high, which makes fault-tolerance difficult. GFS addresses this problem. It is also optimized for Google data-intensive applications, such as search engine or data analysis. These applications have the following characteristics. First, their files are very large, typically several gigabytes, containing many objects such as web documents. Second, workloads consist mainly of read and append operations, while random updates are rare. Read operations consist of large reads of bulk data (e.g., 1 MB) and small random reads (e.g., a few KBs). The append operations are also large and there may be many concurrent clients that append the same file. Third, because workloads consist mainly of large read and append operations, high throughput is more important than low latency.

GFS organizes files as a tree of directories and identifies them by pathnames. It provides a file system interface with traditional file operations (create, open, read, write, close, and delete file) and two additional operations: snapshot and record append. Snapshot allows creating a copy of a file or of a directory tree. Record append allows appending data (the record) to a file by concurrent clients in an efficient way. A record is appended atomically, i.e., as a continuous byte string, at a byte location determined by GFS. This avoids the need for distributed lock management that would be necessary with the traditional write operation (which could be used to append data).

The architecture of GFS is illustrated in Figure 18.6. Files are divided into fixed-size partitions, called *chunks*, of large size, i.e., 64 MB. The cluster nodes consist of GFS clients that provide the GFS interface to applications, chunk servers that store chunks and a single GFS master that maintains file metadata such as namespace, access control information, and chunk placement information. Each chunk has a unique id assigned by the master at creation time and, for reliability reasons, is replicated on at least three chunk servers (in Linux files). To access chunk data, a client must first ask the master for the chunk locations, needed to answer the application file access. Then, using the information returned by the master, the client can request the chunk data to one of the replicas.

This architecture using single master is simple. And since the master is mostly used for locating chunks and does not hold chunk data, it is not a bottleneck. Furthermore, there is no data caching at either clients or chunk servers, since it would not benefit large reads. Another simplification is a relaxed consistency model for concurrent writes and record appends. Thus, the applications must deal with relaxed consistency using techniques such as checkpointing and writing self-validating records. Finally, to keep the system highly available in the face of frequent node failures, GFS relies on fast recovery and replication strategies.

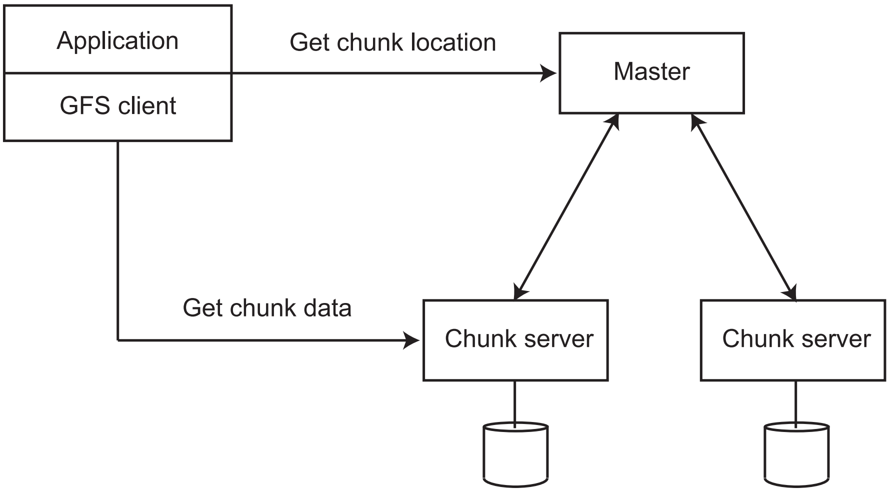

Fig. 18.6 GFS Architecture

18.2.4.2 Distributed Database Management

We can distinguish between two kinds of solutions: online distributed database services and distributed database systems for cloud applications. Online distributed database services such as Amazon SimpleDB and Google Base enable any web user to add and manipulate structured data in a database in a very simple way, without having to define a schema. For instance, SimpleDB provides basic database functionality including scan, filter, join and aggregate operators, caching, replication and transactions, but no complex operators (e.g., union), no query optimizer and no fault-tolerance. Data are structured as (attribute name, value) pairs, all automatically indexed so there is no need for administration. Google Base is a simpler online database service (as a Beta version at the time of this writing) which enables a user to add and retrieve structured data through predefined forms, with predefined attributes (e.g., ingredient for a recipe), thus avoiding the need for schema definition. Data in Google Base can then be searched through other tools, such as the web search engine.

Distributed database systems for cloud applications emphasize scalability, fault-tolerance and availability, sometimes at the expense of consistency or ease of development. We illustrate this approach with two popular solutions: Google Bigtable and Yahoo! PNUTS.

Bigtable.

Bigtable is a database storage system for a shared-nothing cluster [Chang et al., 2008]. It uses GFS for storing structured data in distributed files, which provides fault-tolerance and availability. It also uses a form of dynamic data partitioning for scalability. And like GFS, it is used by popular Google applications, such as Google Earth, Google Analytics and Orkut. There are also open source implementations of Bigtable, such as Hadoop Hbase, which runs on HDFS.

Bigtable supports a simple data model that resembles the relational model, with multi-valued, timestamped attributes. We briefly describe this model as it is the basis for Bigtable implementation that combines aspects of row-store and column-store DBMS. We use the terminology of the original proposal [Chang et al., 2008], in particular, the basic terms "row" and "column" (instead of tuple and attribute). However, for consistency with the concepts we have used so far, we present the Bigtable data model as a slightly extended relational model[1]. Each row in a table (or Bigtable) is uniquely identified by a *row key*, which is an arbitrary string (of up to 64KB in the original system). Thus, a row key is like a mono-attribute key in a relation. A more original concept is that of a *column family* which is a set of columns (of the same type), each identified by a *column key*. A column family is a unit of access control and compression. The syntax for naming column keys is `family:qualifier`. The column family name is like a relation attribute name. The qualifier is like a relation attribute value, but used as a name as part of the column key to represent a single data item. This allows the equivalent of multi-valued attributes within a relation, but with the capability of naming attribute values. In addition, the data identified by a column key within a row can have multiple versions, each identified by a timestamp (a 64 bit integer).

Figure 18.7 shows an example a row in a Bigtable, as a relational style representation of the example [Chang et al., 2008]. The row key is a reverse URL. The Contents:column family has only one column key that represents the web page contents, with two versions (at timestamps t_1 and t_5). The Language:family has also only one column key that represents the web page language, with one version. The Anchor: column family has two column keys, i.e., Anchor:inria.fr and Anchor:uwaterloo.ca, which represent two anchors. The anchor source site name (e.g., inria.fr) is used as qualifier and the link text as value.

Bigtable provides a basic API for defining and manipulating tables, within a programming language such as C++. The API offers various operators to write and update values, and to iterate over subsets of data, produced by a scan operator. There are various ways to restrict the rows, columns and timestamps produced by a scan, as in a relational select operator. However, there are no complex operators such as join or union, which need to be programmed using the scan operator. Transactional atomicity is supported for single row updates only.

To store a table in GFS, Bigtable uses range partitioning on the row key. Each table is divided into partitions called *tablets*, each corresponding to a row range. Partitioning is dynamic, starting with one tablet (the entire table range) that is subsequently split into multiple tablets as the table grows. To locate the (user) tablets in GFS, Bigtable uses a metadata table, which is itself partitioned in metadata tablets, with a single root tablet stored at a master server, similar to GFSs master. In addition

[1] In the original proposal, a Bigtable is defined as a multidimensional map, indexed by a row key, a column key and a timestamp, each cell of the map being a single value (a string).

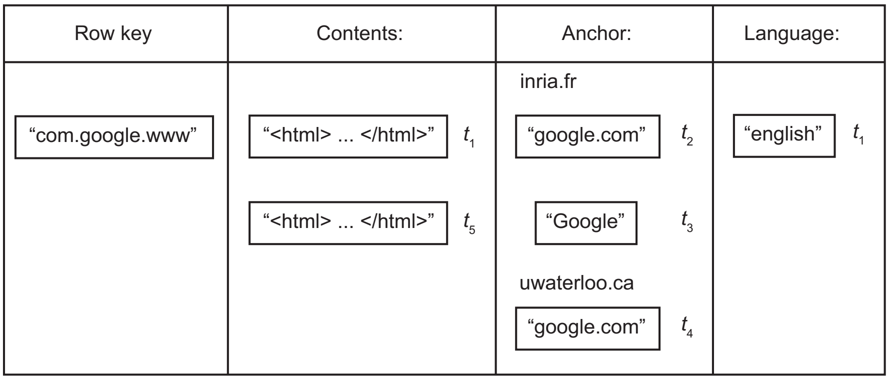

Fig. 18.7 Example of a Bigtable Row

to exploiting GFS for scalability and availability, Bigtable uses various techniques to optimize data access and minimize the number of disk accesses, such as compression of column families, grouping of column families with high locality of access and aggressive caching of metadata information by clients.

PNUTS.

PNUTS is a parallel and distributed database system for Yahoo!'s cloud applications [Cooper et al., 2008]. It is designed to serve web applications, which typically do not need complex queries, but require good response time, scalability and high availability and can tolerate relaxed consistency guarantees for replicated data. PNUTS is used internally at Yahoo! for various applications such as user database, social networks, content metadata management and shopping listings management.

PNUTS supports the basic relational data model, with tables of flat records. However, arbitrary structures are allowed within attributes of Binary Long Object (Blob) type. Schemas are flexible as new attributes can be added at any time even though the table is being queried or updated, and records need not have values for all attributes. PNUTS provides a simple query language with selection and projection on a single relation. Updates and deletes must specify the primary key.

PNUTS provides a replica consistency model that is between strong consistency and eventual consistency (see Chapter 13 for detailed definitions). This model is motivated by the fact that web applications typically manipulate only one record at a time, but different records may be used under different geographic locations. Thus, PNUTS proposes *per-record timeline consistency*, which guarantees that all replicas of a given record apply all updates to the record in the same order. Using this consistency model, PNUTS supports several API operations with different guarantees. For instance, **Read-any** returns a possibly stale version of the record; **Read-latest** returns the latest copy of the record; **Write** performs a single atomic write operation.

Database tables are horizontally partitioned into tablets, through either range partitioning or hashing, which are distributed across many servers in a cluster (at a site). Furthermore, sites in different geographical regions maintain a complete copy of the system and of each table. An original aspect is the use of a publish/subscribe mechanism, with guaranteed delivery, for both reliability and replication. This avoids the need to keep a traditional database log as the publish/subscribe mechanism is used to replay lost updates.

18.2.4.3 Parallel Data Processing

We illustrate parallel data processing in the cloud with MapReduce, a popular programming framework for processing and generating large datasets [Dean and Ghemawat, 2004]. MapReduce was initially developed by Google as a proprietary product to process large amounts of unstructured or semi-structured data, such as web documents and logs of web page requests, on large shared-nothing clusters of commodity nodes and produce various kinds of data such as inverted indices or URL access frequencies. Different implementations of MapReduce are now available such as Amazon MapReduce (as a cloud service) or Hadoop MapReduce (as open source software).

MapReduce enables programmers to express in a simple, functional style their computations on large data sets and hides the details of parallel data processing, load balancing and fault-tolerance. The programming model includes only two operations, *map* and *reduce*, which we can find in many functional programming languages such as Lisp and ML. The Map operation is applied to each record in the input data set to compute one or more intermediate (key,value) pairs. The Reduce operation is applied to all the values that share the same unique key in order to compute a combined result. Since they work on independent inputs, Map and Reduce can be automatically processed in parallel, on different data partitions using many cluster nodes.

Figure 18.8 gives an overview of MapReduce execution in a cluster. There is one master node (not shown in the figure) in the cluster that assigns Map and Reduce tasks to cluster nodes, i.e., Map and Reduce nodes. The input data set is first automatically split into a number of partitions, each being processed by a different Map node that applies the Map operation to each input record to compute intermediate (key,value) pairs. The intermediate result is divided into n partitions, using a partitioning function applied to the key (e.g., hash(key) mod n). Map nodes periodically write to disk their intermediate data into n regions by applying the partitioning function and indicate the region locations to the master. Reduce nodes are assigned by the master to work on one or more partitions. Each Reduce node first reads the partitions from the corresponding regions on the Map nodes, disks, and groups the values by intermediate key, using sorting. Then, for each unique key and group of values, it calls the user Reduce operation to compute a final result that is written in the output data set.

As in the original description of MapReduce [Dean and Ghemawat, 2004], the favorite examples deal with sets of documents, e.g., counting the occurrences of each word in each document, or matching a given pattern in each document. However,

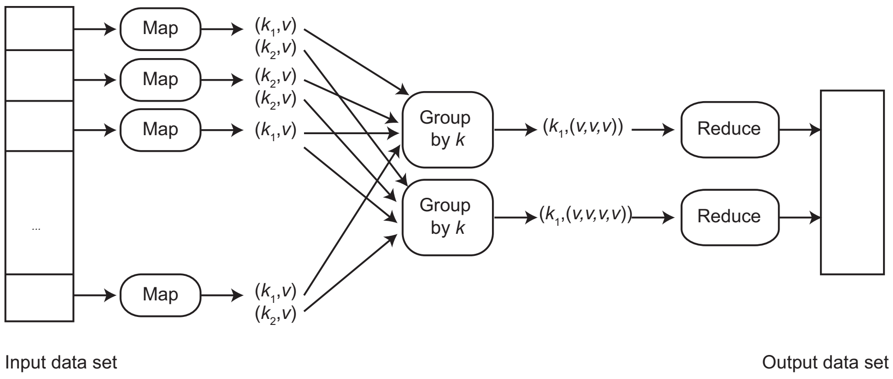

Fig. 18.8 Overview of MapReduce Execution

MapReduce can also be used to process relational data, as in the following example of a Group By select query on a single relation.

Example 18.3. Let us consider relation EMP(ENAME, TITLE, CITY) and the following SQL query that returns for each city, the number of employees whose name is "Smith".

```
SELECT   CITY, COUNT(*)
FROM     EMP
WHERE    ENAME LIKE "%Smith"
GROUP BY CITY
```

Processing this query with MapReduce can be done with the following Map and Reduce functions (which we give in pseudo code).

```
Map (Input (TID,emp), Output: (CITY,1))
    if emp.ENAME like "%Smith" return (CITY,1)
Reduce (Input (CITY,list(1)), Output: (CITY,SUM(list(1)))
    return (CITY,SUM(1*))
```

Map is applied in parallel to every tuple in EMP. It takes one pair (TID,emp), where the key is the EMP tuple identifier (TID) and the value the EMP tuple, and, if applicable, returns one pair (CITY,1). Note that the parsing of the tuple format to extract attributes needs to be done by the Map function. Then all (CITY,1) pairs with the same CITY are grouped together and a pair (CITY,list(1)) is created for each CITY. Reduce is then applied in parallel to compute the count for each CITY and produce the result of the query. ♦

Fault-tolerance is important as there may be many nodes executing Map and Reduce operations. Input and output data are stored in GFS that already provides high fault-tolerance. Furthermore, all intermediate data are written to disk that helps checkpointing Map operations, and thus provides tolerance to soft failures. However, if one Map node or Reduce node fails during execution (hard failure), the task can be scheduled by the master onto other nodes. It may also be necessary to re-execute completed Map tasks, since the input data on the failed node disk is inaccessible. Overall, fault-tolerance is fine-grained and well suited for large jobs.

MapReduce has been extensively used both within Google and outside, with the Hadoop open source implementation, for many various applications including text processing, machine learning, and graph processing on very large data sets. The often cited advantages of MapReduce are its ability to express various (even complicated) Map and Reduce functions, and its extreme scalability and fault-tolerance. However, the comparison of MapReduce with parallel DBMSs in terms of performance has been the subject of debate between their respective proponents [Stonebraker et al., 2010; Dean and Ghemawat, 2010]. A performance comparison of Hadoop MapReduce and two parallel DBMSs – one row-store and one column-store DBMS – using a benchmark of three queries (a grep query, an aggregation query with a group by clause on a web log, and a complex join of two tables with aggregation and filtering) shows that, once the data has been loaded, the DBMSs are significantly faster, but loading data is very time consuming for the DBMSs [Pavlo et al., 2009]. The study also suggests that MapReduce is less efficient than DBMSs, because it performs repetitive format parsing and does not exploit pipelining and indices. It has been argued that a differentiation needs to be made between the MapReduce model and its implementations, which could be well improved, e.g., by exploiting indices [Dean and Ghemawat, 2010]. Another observation is that MapReduce and parallel DBMSs are complementary as MapReduce could be used to extract-transform-load data in a DBMS for more complex OLAP [Stonebraker et al., 2010].

18.3 Conclusion

In this chapter, we discussed two topics that are currently receiving considerable attention – data stream management, and cloud data management. Both of these have the potential to make considerable impact on distributed data management, but they are still not fully matured and require more research.

Data stream management addresses the requirements of a class of applications that produce data continuously. These systems require a shift in emphasis from traditional DBMSs in that they deal with data that is transient and queries that are (generally) persistent. Thus, they require new solutions and approaches. We discussed the main tenets of data stream management systems (DSMSs) in this chapter. The main challenge in data stream management is that data are produced continually, so it is not possible to store them for processing, as is typically done in traditional DBMSs. This requires unblocking operations, and online algorithms that sometimes have to deal with high data rates. The abstract models, language issues, and windowed query processing of streams are relatively well understood. However, there are a number of interesting research directions including the following:

- **Scaling with data rates.** Some data streams are relatively slow, while others have very high data rates. It is not clear if the strategies that have been developed for processing queries work on the wide range of stream rates. It is probably the case that special processing techniques need to be developed for different classes of streams based on their data rates.

- **Distributed stream processing.** Although there has been some amount of work in considering processing streams in a distributed fashion, most of the existing works consider a single processing site. Distribution, as is usually the case, poses new challenges but also new opportunities that are worth exploring.
- **Stream data warehouses.** Stream data warehouses combine the challenges of standard data warehouses and data streams. This is an area that has recently started to receive attention (e.g., [Golab et al., 2009; Polyzotis et al., 2008]), but there are still many problems that require attention, including update scheduling strategies for optimizing various objectives, and monitoring data consistency and quality as new data arrive [Golab and Özsu, 2010].
- **Uncertain data streams.** In many applications that generate streaming data, there may be uncertainty in the data values. For example, sensors may be faulty and generate data that are not accurate, certain observations may be uncertain, etc. The processing of queries over uncertain data streams poses significant challenges that are still open.

One of the main challenges of cloud data management is to provide ease of programming, consistency, scalability and elasticity at the same time, over cloud data. Current solutions have been quite successful but developed with specific, relatively simple applications in mind. In particular, they have sacrificed consistency and ease of programming for the sake of scalability. This has resulted in a pervasive approach relying on data partitioning and forcing applications to access data partitions individually, with a loss of consistency guarantees across data partitions. As the need to support tighter consistency requirements, e.g., for updating multiple tuples in one or more tables, increases, cloud application developers will be faced with a very difficult problem: providing isolation and atomicity across data partitions through careful engineering. We believe that new solutions are needed that capitalize on the principles of distributed and parallel database systems to raise the level of consistency and abstraction, while retaining the scalability and simplicity advantages of current solutions. Parallel database management techniques such as pipelining, indices and optimization should also be useful to improve the performance of MapReduce-like systems and support more complex data analysis applications. In the context of large-scale shared-nothing clusters, where node failures become the norm rather than the exception, another important problem remains to deal with the trade-off between query performance and fault-tolerance. P2P techniques that do not require centralized query execution control by a master node could also be useful there. Some promising research directions for cloud data management include the following:

- **Declarative programming languages.** Programming large-scale, distributed data management software such as MapReduce remains very hard. One promising solution proposed in the BOOM project [Alvaro et al., 2010] is to adopt a data centric declarative programming language, based on the Overlog data language, in order to improve ease of development and program correctness without sacrificing performance.

- **Autonomic data management.** Self-management of the data by the cloud will be critical to support large numbers of users with no database expertise. Modern database systems already provide good self-administration, self-tuning and self-repairing capabilities which ease application deployment and evolution. However, extending these capabilities to the scale of a cloud is hard. In particular, one problem is the automatic management of replication (definition, allocation, refreshment) to deal with load variations [Doherty and Hurley, 2007].
- **Data security and privacy.** Data security and access control in a cloud typically rely on user authentication and secured communication protocols to exchange encrypted data. However, the semi-open nature of a cloud makes security and privacy a major challenge since users may not trust the providers servers. Thus, the ability to perform relational-like operators directly on encrypted data at the cloud is important [Abadi, 2009]. In some applications, it is important that data privacy be preserved, using high-level mechanisms such as those of Hyppocratic databases [Agrawal et al., 2002].
- **Green data management.** One major problem for large-scale clouds is the energy cost. Harizopoulos et al. [2009] argue that data management techniques will be key in optimizing for energy efficiency. However, current data management techniques for the cloud have focused on scalability and performance, and must be significantly revisited to account for energy costs in query optimization, data structures and algorithms.

Finally there are problems in the intersection of data stream processing and cloud computing. Given the steady increase in data stream volumes, the need to process massive data flows in a scalable way is becoming important. Thus, the potential scalability advantage of a cloud can be exploited for data stream management as in Streamcloud [Gulisano et al., 2010]. This requires new strategies to parallelize continuous queries. And deadling with various trade-offs.

18.4 Bibliographic Notes

Data streams have received a lot of attention in recent years, so the literature on the topic is extensive. Good early overviews are given in [Babcock et al., 2002; Golab and Özsu, 2003a]. A more recent edited volume [Aggarwal, 2007] includes a number of articles on various aspects of these systems. An volume [Golab and Özsu, 2010] gives a full treatment of many of the issues that are discussed here. Mining data streams is reviewed in [Gaber et al., 2005] and issues in mining data streams with underlying distribution changes is discussed in [Hulten et al., 2001].

Our discussion of data stream systems follows [Golab and Özsu, 2003a], Chapter 2 of [Golab, 2006] and [Golab and Özsu, 2010]. The discussion on mining data streams borrows from Chapter 2 of [Tao, 2010].

Cloud computing has recently gained a lot of attention from the professional press as a new platform for enterprise and personal computing (see [Cusumano, 2010] for a good discussion of the trend). However, the research literature on cloud computing in general, and cloud data management in particular, is rather small, but as the number of international conferences and workshops grow, this should change quickly to become a major research domain. Our cloud taxonomy in Section 18.2.1 is based on our compilation of many professional articles and white papers. The discussion on grid computing in Section 18.2.2 is based on [Atkinson et al., 2005; Pacitti et al., 2007b]. The section on data management in the cloud (Section 18.2.4) has been inspired by several keynotes on the topic, e.g., [Ramakrishnan, 2009]. The technical details can be found in the research papers on GFS [Ghemawat et al., 2003], Bigtable [Chang et al., 2008], PNUTS [Cooper et al., 2008] and MapReduce [Dean and Ghemawat, 2004]. The discussion of MapReduce versus parallel DBMS can be found in [Stonebraker et al., 2010; Dean and Ghemawat, 2010].