SFL: A Structured Dataflow Language Based on SQL and FP

Qiming Chen and Meichun Hsu

HP Labs
Palo Alto, California, USA
Hewlett Packard Co.
{qiming.chen,meichun.hsu}@hp.com

Abstract. SFL (pronounced as Sea-Flow) is an analytics system that supports a declarative language that extends SQL for specifying the dataflow of data-intensive analytics. The extended SQL language is motivated by providing a top-level representation of the converged platform for analytics and data management. Due to fast data access and reduced data transfer, such convergence has become the key to speed up and scale up data intensive BI applications.

A SFL query is constructed from conventional queries in terms of Function Forms (FFs). While a conventional SQL query represents a dataflow tree, a SFL query represents a more general dataflow graph. We support SFL query execution by tightly integrating it with the evaluation of its component queries to minimize the overhead of data retrieval, copying, moving and buffering, which actually turns a query engine to a generalized dataflow engine. The experimental results based on a prototype built by extending the PostgreSQL engine are discussed.

1 Introduction

A Business Intelligence (BI) application often form a dataflow process from collected data to derived information and decision. With the current technology, BI system software architecture generally separates the BI analytics layer (BI applications and tools) from the data management layer (DBMS or Data Warehouses), where applications are coded as database client programs in C, Java, T-SQL, P/L-SQL, etc, which invoke SQL statement for data retrieval. As the steep increase in the amount, the data transferred between the analytics platform and the database platform has become the scalability and performance bottleneck of BI applications.

Converging data-intensive analytics computation into the DB engine is the key to address these problems [2,4]. One option for the next generation BI system is to have the data intensive part of analytics executed inside the DB engine, which implies that the application dataflow and database access should be specified by a single, integrated declarative language. It is argued that SQL is a reasonable candidate for such a language.

In order to express analytics operations which are beyond the standard relational database operations, we rely on User Defined Functions (UDFs) [3]. However, a SQL

P. García Bringas et al. (Eds.): DEXA 2010, Part I, LNCS 6261, pp. 306–314, 2010.
© Springer-Verlag Berlin Heidelberg 2010

query is limited to express, and the existing query engine is limited to orchestrate the tree-structured dataflow; to handle complex, graph-structured inter-query dataflow inside the database system, extending the query language as well as the query engine is required.

To *declaratively* express complex dataflow graph, we introduce certain *construction primitives* and Function Forms (FFs) from the functional programming language FP [1] into the SQL framework. The extended SQL language uses these primitives and FFs to glue queries and UDFs based on a calculus of queries. We use this extended language as the top-level representation of a converged platform for analytics and data management called SFL (pronounced as Sea-Flow) platform. While a conventional SQL query represents a dataflow tree, a SFL query represents a more general dataflow graph declaratively. This extension has two important features: first, the extended language specifies complex application data flows using queries invoking functions *declaratively* rather than imperatively, thus isolates much of the complexity of data streaming into a well-understood system abstraction; second the same data object can "flow" to multiple operations without copying, repeated retrieval from database or duplicated query evaluations, which supports the justification of pushing analytics down to the query engine rather than at the client level.

The SFL query is supported in the SFL platform with an query engine with the functionalities that orchestrate actions (queries) interactively with the underlying query processing. Our experience shows the value of the proposed approach in multiple dimensions: modeling capability, efficiency, as well as usability; all these represent a paradigm shift from traditional BI in converging data-intensive analytics and data management.

2 SFL Language Framework

A SFL system is used to combine queries and user defined functions for representing application dataflow graphs. In the following we introduce the operators – functional forms, and operands – query functions and user-defined relational operator functions (RVFs), of the SFL language framework.

First, we rely on UDFs to extend the action capability of the database engine. In order to handle operations applied to a set of tuples (a relation) rather than a single tuple, we have introduced the kind of UDFs with input as a list of relations and with return value as a relation, called Relation Valued Functions (RVFs) [3]. RVFs can be integrated to SQL queries and treated as relational operators or relational data objects.

Next, we distinguish the notion of *Query Function* from the notion of *Query Variable*. A query variable is just a query such as

```
SELECT * FROM Orders, Customers WHERE Orders.customer_id = Customers.id;
```

that is bound to actual relations such as the above Orders and Customers. A query variable can be viewed as a relation data object, say Q_v, denoting the query result.

A query function is a function applied to a sequence of parameter relations. For instance, the query function corresponding to the above query can be expressed as

Q_f := SELECT * FROM $1, $2 WHERE $1.customer_id = $2.id;

Then applying Q_f to a sequence of relations < Orders, Customers> with matched schemas, is expressed by

Q_f : < Orders, Customers> → Q_v

The major constraint of query functions is *schema-preservation*, i.e. the schemas of the parameter relations must match the query function. It is obvious that the above query function is not applicable to arbitrary relations.

Further, we introduce the notions of constructive-primitive and functional form. A constructive-primitive expresses the path of function application, or the path of dataflow. For example, applying a *Selector* Function $\$i$ to a sequence of objects $<r_1, \ldots, r_n>$ returns r_i; applying the *Identity* Function *id* to object *r* returns *r* itself.

A functional form (or function combining form), FF, is an expression denoting a function; that function depends on the functions which are the parameters of the expression. Thus, for example, if f and g are RVFs, then $f \bullet g$ is a functional form denoting a new function such that, for a relation r, $(f \bullet g)$:r = f:(g:r) provided that r matches the input schema of g, and g:r matches the input schema of f.

The dataflow shown in Fig. 1(a), where Q_f is a query function "SELECT * FROM \$1, \$2", f_1 and f_2 are RVFs with inputs r_1 and r_2, can be expressed with the use of the *constructive function combining form* denoted as [ ], which constructs a list from the content in the bracket:

$Q_f \bullet [f_1 \bullet \$1,\ f_2 \bullet \$2] : <r_1, r_2>$

that denotes $Q_f\,(f_1(r_1), f_2(r_2))$. Note that this statement could be expanded to

"SELECT * FROM $f_1(r_1), f_2(r_2)$".

For the dataflow in Fig.1 (b) where r_2 needs to feed into both f_1 and f_2, the dataflow is forked. This dataflow is written as:

$Q_f \bullet [f_1 \bullet \text{id},\ f_2 \bullet \$2] : <r_1, r_2>$

where *id* and *\$2* are primitive constructors applied to the argument list. In this case, *id* applied to $<r_1, r_2>$ returns $<r_1, r_2>$ itself, while *\$2* applied to $<r_1, r_2>$ returns r_2, and thus Q_f applies to (or composes with) results of f_1 and f_2, and f_1 applies to *both* $<r_1, r_2>$, while f_2 applies to second argument of $<r_1, r_2>$. We see that forking cannot be accomplished without extension to SQL.

Further, suppose we need to express the dataflow shown in Fig 1(c) where Q_f has three input objects, and r_1 needs to feed into both f_1 and Q_f. Then the query statement would be

$Q_f \bullet [\$1, ,f_1 \bullet \text{id},\ f_2 \bullet \$2] : <r_1, r_2>$

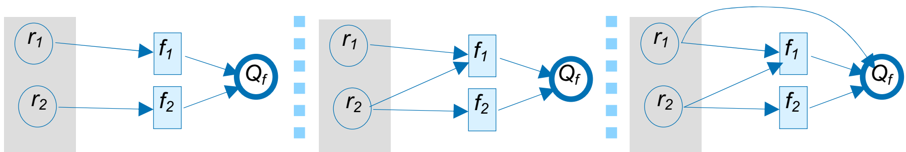

Fig. 1. (a) dataflow without fork (b) dataflow with fork (c) dataflow with fork & crossover

In general. a SFL system is founded on the use of a fixed set of FFs for combining query functions (a query can invoke RVFs). These, plus simple definitions, provide the simple means of building new functions from existing ones; they use no variables or substitution rules, and they become the operations of an associated algebra of

queries. All the functions of a SFL system are of one type: they map relations into relations; and they are the operands of FFs. Applying an FF to functions denotes a new function; and applying that function to relations denotes a *SFL query*. A SFL query has the expressive power for specifying a dataflow graph, in a way not possible by a regular query. Specifically, a regular SQL query represents a tree-structured dataflow, while a SFL query can further represent graph structured dataflow. A more formal description can be found in [2].

3 An Example

In this section we shall illustrate how to express the dataflow scheme of K-Means clustering using a single SFL statement and discuss our experimental results. The *k*-means algorithm illustrated in Fig 2 is an algorithm to cluster *n* objects based on attributes into *k* partitions, $k < n$. It is similar to the expectation-maximization algorithm for mixtures of Gaussians in that they both attempt to find the centers of natural clusters in the data. It assumes that the object attributes form a vector space.

The K-Means computation used in this example has two original input relations, *Points* and *Centers*, and two relational operations, *cluster* and *check*. The *cluster* operation generates a new set of centers to be compared with the existing centers by the *check* operation; if they have not converged enough, the *cluster* will be re-run iteratively with the same *Points* data and the new centers. Note that SQL is unable to express the iteration.

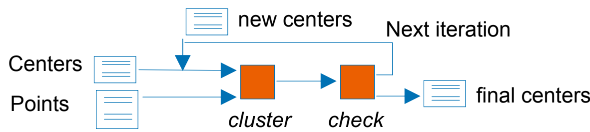

Fig. 2. K-Means clustering

Let us consider the K-Means calculation on two dimensional geographic points. The original Points data are stored in relation Points [x, y, …], and the Centers data are stored in relation Centers [cid, x, y, …]). Two RVFs are involved. The RVF

cluster_rvf : <Points, Centers> → Centers'

is used to derive a new set of centers, i.e. a new instance of relation Centers, from relations Points and Centers in a single iteration, in the following two steps:

- the first step is for each point in relation Points to compute its distances to all centers in relation Centers and assign its membership to the closest one, resulting an intermediate relation for new centers [x, y, cid];
- the second step is to re-compute the set of new centers based on the average location of member points.

After an iteration, the newly derived centers (in relation Centers') are compared to the old ones (in relation Centers) by another RVF

check_rvf : <Centers', Centers> → {T; F}

for checking the convergence of the sets of new and old centers to determine whether to terminate the K-Means computation or to launch the next iteration using the current centers, as well as the original points as the input data.

Our goal is to define a SFL query *kmeans* : <Points, Centers>

that derives, in multiple iterations, the Centers of the clusters with minimal total intra-cluster variance, from the initial Points and Centers relations, towards the final instance of relation Centers. During the *kmeans* computation, the relation Centers is updated in each iteration, but the relation Points remains the same. A key requirement is to avoid repeated retrieval of either the Centers relation or the Points relation from the database, which should be explicitly expressible at the language presentation level.

The function *kmeans* is defined by the following

clustering := [$1, $2, *cluster_rvf*];
iterating := (*check_rvf* •[$2, $3]→$3; *iterating* • *clustering* • [$1, $3]) ;
kmeans := *iterating* • *clustering*;

Applying function *kmeans* to the points and the initial centers for generating the converged center positions is expressed by the SFL query

kmeans : <Points, Centers>

The execution of SFL query *kmeans* : <Points, Centers> is depicted in Fig 6. As described later, the different versions of the instances of relation Centers output from *cluster_rvf* in multiple iterations, actually occupy the same memory space.

4 Implementation Issues

Introducing SFL system aims to support general graph structured dataflow. From the implementation perspective, the key point consists in controlling the sharing and the buffering of data with desired life-span, in order to ensure that data is “piped” in the data flow rather than unnecessarily copied, repeatedly retrieved or duplicated evaluated.

We extended the PostgreSQL engine to support RVFs for integrating applications into queries, and to support SFL queries for expressing application data flows. We have extended the query engine by providing the RVF Manager and the SFL Manager. As we have reported the support to RVF in [3], in this paper we focus on the implementation of SFL Manager.

4.1 SFL Query Memory Context

A SFL query invokes queries during its execution. The memory context of a SFL query execution is accessible to the invoked queries. Such memory sharing allows the data passed in or returned from an invoked query to be referenced without copying, and carried across multiple queries without repeated derivation.

When a SFL query, say, Q_{ff}, is to be executed, an instance of it is created with a memory context for holding the relation data retrieved from the database or generated by the component queries. Based on PostgreSQL internal, the common data structure

for holding these relation data is called *tuple-store*. A tuple-store is an in-memory data structure with memory overflow handled automatically by the underlying query executor (e.g. expanded to temporal files). The system internal query facility, such as PostgreSQL SPI (Server Program Interface), is extended with tuple-store access-method, and the handles of multiple related queries are made sharable at the SFL layer.

During the execution of Q_{ff}, all the component query results are kept in tuple-stores. A query invoked by Q_{ff}, say q, does not copy input data into its own memory context; instead the data is passed in by reference. More concretely, the memory context of Q_{ff}, say MC and that of q, say, MC_q have the following relationship.

- The life-span of MC covers that of MC_q.
- The tuple-stores in MC are used by q as required; that is, the input relations of q are directly obtained from MC with pass-by-reference; the output relation of q is returned to MC.

Specifically, associated with a SFL query instance is a handle structure for its execution environment, h_{env}, with a pointer to its memory context handle, say h_{mc}. Associated with a component query is also a handle for holding its own execution information, and that handle has a pointer to h_{mc} for accessing and updating the data flowing through the SFL query process.

The mechanisms for PostgreSQL memory management, including the mechanism for managing shared buffer pool, are leveraged. The cached data are kept in the memory resided tuple-stores. An overflow tuple-store can be backed up by a file automatically and transparently with the same access APIs. This ensure our system survive with large initial data. In fact, different from data warehousing, the basic principle in dealing with dataflow applications is to keep data “moving” rather than “staying”, therefore most dataflow applications are designed to fit in moderate on-the-flow buffer size.

4.2 SFL Query Data Flows

The functionality difference of a query/RVF and a constructive primitive has been explained above. Here we examine it from the dataflow point of view.

- A data object R flowing through a query or RVF means that the query or RVF produces an output relation out of R;
- R flowing through a constructive primitive G such as *id* or *$1* means that R is to remain referenced, and thus active; what actually passed through G is the reference, or pointer if handled in a single query engine, of R; such reference-flow mechanism ensures that R is not to be copied; and based on the *reference-counting* mechanism, garbage-collected when not being referenced anymore (following a common rule, we do not distinguish the value and the reference of a primitive typed object such as an integer used as a function argument).

To be concrete, let us refer to Fig 3 and consider two data buffer pools $POOL_1$ and $POOL_2$, i.e. the memory closures for *clustering* and *iterating*. The original relations *Points* (P) and *Centers* (C) retrieved from the database are buffered at $POOL_1$, the intermediate relation new centers derived by RVF *luster_rvf* in each iteration is buffered at $POOL_2$. The data flows of P and C can be described as below.

- The flow of Points data P:
  - at $POOL_1$, P is referenced by functions *cluster_rvf* and *clustering* thus active;
  - at $POOL_2$, P is still referenced by function *iterating* thus active;
  - if a new iteration is launched, P is not reproduced since it is then referenced by *clustering* again as its *$1* argument, thus kept active rather than re-produced;
  - therefore, P is buffered only once after retrieved from the database, with its reference "flows".
- The flow of Centers data C:
  - at $POOL_1$, C is referenced by functions *cluster_rvf* and *check_rvf*, thus active;
  - a new version of C, C', is generated by *cluster_rvf* at $POOL_2$;
  - after *check_rvf* is executed, C is not referenced by anyone so will be garbage collected or refreshed;
  - if a new iteration is launched, the data of C' flow to $POOL_1$ to refresh the buffer.

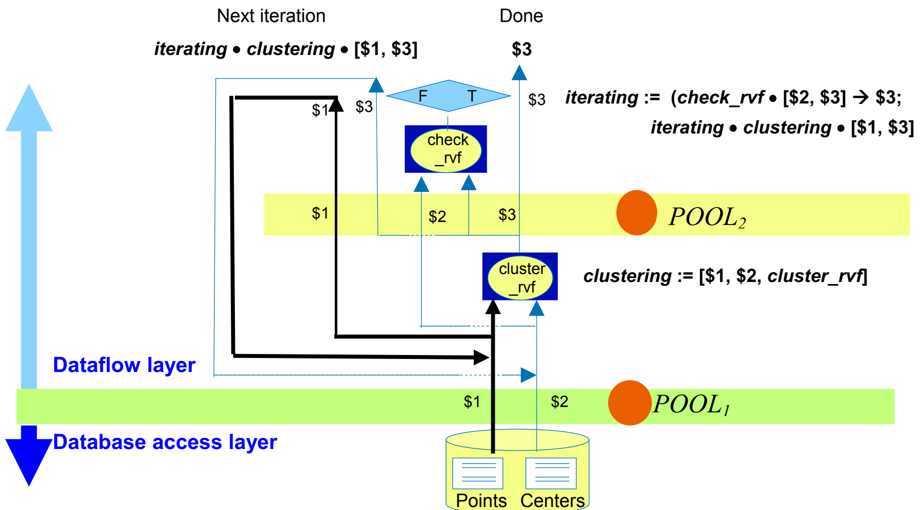

Fig. 3. K-Means dataflow without repeated querying and database retrieval

4.3 SFL Query Execution

To support SFL, we have plugged a preliminary Structured Dataflow Manager (SFM) in the PostgreSQL query engine, with two major components:

- FH – i.e. Flow Handler for creating and managing SFL instances, and for scheduling operations;
- QH – i.e. Query Handler for launching queries (possibly involving RVFs) using the PostgreSQL internal SPI query facility extended with tuple-store access.

With the FH and QH, a SFL query is scheduled in the following way.

- Based on the template of a SFL query, the FH creates an SFL query process *instance* and derives one control-sequence of its operations, resulting in an execution plan.
- The FH creates the start operation instances based on the control-sequence.
- For running every operation, the FH identifies the incoming tuple-stores, identifies or creates the outgoing ones, generates a corresponding *operation-item*, and puts it to the *operation queue*. At the minimum, an operation item includes the ID of the containing query-instance (i.e. the SFL query instance), the operation-instance-ID, the query (may involve RVFs) of this operation, and the references to incoming and outgoing tuple-stores (a reference is a key name, not a physical pointer).
- The operation items, once put in the operation queue, can be executed in any order, and in fact they are de-queued in pipeline and executed by individual, time-overlapping threads.
- The QH runs as a separate thread of the FH, it de-queues the operation items one by one; for each item, it launches a thread to execute the query associated with that item, using the high-efficient PostgreSQL internal SPI facility (extended with tuple-store access), then puts the returned query result into the outgoing tuple-stores, and sends a return value, *rv*, to the FH. The return value is a message that contains, at the minimum, the process-instance-ID, the operation-instance-ID, the query execution status returned from SPI, and the references to outgoing tuple-stores.
- Upon receipt of an *rv*, the FH updates the corresponding operation instance and process instance, checks the control-sequence and triggering condition or timer, then selects the next eligible operation or operations to run, rolling forward the process instance, towards the end of it.

The SFM is actually a "*middleware inside the query engine*" for handling multiple SFL queries with interleaved operation executions. Its scheduling mechanism is similar to that of a BPM; however, the running environments of a business process and its tasks are in general isolated, but the running environments of a SFL query and a component query is joined.

5 Conclusions

To effectively handle the scale of analytical tasks in an era of information explosion, pushing data-intensive analytics down to the database engine is the key. In this work we tackled two problems for converging analytics and data management: first, integrating general analytic operations into SQL queries, and second, extend SQL framework to express general graph structured dataflow.

We support SFL by building a super-query processing container inside the database engine that deals with data buffering, dataflow, control-flow and function orchestration. During execution, the memory context of a SFL query is tightly integrated with that of the queries it invokes, which effectively eliminates the overhead for data copying and duplicated retrieval or derivation.

The advantages of SFL lies in its expressive power for specifying complex dataflow, as well as in its simplicity, as it uses only the most elementary fixed naming system (naming a query) with a simple fixed rule of substituting a query for its name. Most importantly, they treat names as functions that can be combined with other functions without special treatment.

References

[1] Backus, J.: Can Programming Be Liberated from the von. Neumann Style? A. Functional. Style and Its. Algebra of Programs. ACM Turing award lecture (1977)
[2] Chen, Q., Hsu, M.: Cooperating SQL Dataflow Processes for In-DB Analytics. In: CoopIS'09 (2009)
[3] Chen, Q., Hsu, M., Liu, R.: Extend UDF Technology for Integrated Analytics. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 256–270. Springer, Heidelberg (2009)
[4] DeWitt, D.J., Paulson, E., Robinson, E., Naughton, J., Royalty, J., Shankar, S., Krioukov, A.: Clustera: An Integrated Computation And Data Management System. In: VLDB 2008 (2008)