Cooperating SQL Dataflow Processes for In-DB Analytics

Qiming Chen and Meichun Hsu

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

Abstract. Pushing data-intensive analytics down to database engines is the key to high-performance and secured execution; however, the existent SQL framework is unable to express general graph-based dataflow processes, and unable to orchestrate multiple dataflow processes with inter-operation data dependencies.

In this work we extend SQL to Functional Form-SQL (FF-SQL) based on a calculus of queries, to *declaratively* express complex dataflow graphs. A FF-SQL query is constructed from conventional queries using Function Forms (FFs). While a conventional SQL query represents a dataflow tree, a FF-SQL query represents a more general dataflow graph. Further, with FF-SQL, a group of SQL dataflow processes with data dependency among their operations can be specified as a single, integrated FF-SQL definition, and executed cooperatively inside the database engine without repeated data retrieval, duplicated computation and unnecessary data copying. A novel extension to the PostgreSQL query engine is made to support FF-SQL dataflow processes.

1 Introduction

Executing data-intensive BI analytics inside the database engine can provide benefits in scalability, performance and security [3,5]. However, this approach is not yet generally applicable since the SQL language and SQL engine lack the capability to express a general graph structured dataflow process (beyond a query tree), and to orchestrate multiple intra-process and inter-process operations for sharing database access and query evaluation results. With the existent SQL framework, a single SQL query can only express tree-structured operations with coincident dataflows and control-flows. The result of a (sub)query can only be delivered to a single parent operation; in case it is requested by multiple operations, the query, must be evaluated multiple times. These limit the SQL framework in supporting graph-structured dataflows at both language and implementation levels. In this paper we present our solutions to these problems.

Let us consider 3 applications that use star-joins of a fact table and multiple dimension tables, as shown in Fig. 1.

With the existing SQL framework, the fork of the star-join result to multiple destination operations is not possible; instead, multiple separate queries must perform the star-join repeatedly; as for analytical computation, if we were to push *clustering* into the database engine, SQL is unable to express the iteration and the cache of the *customer feature vectors* across iterations without repeated data loading or derivation.

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 389–397, 2009.

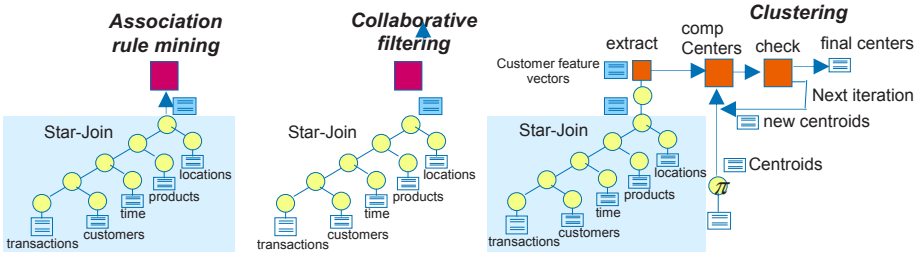

Fig. 1. SQL is unable to express the "share" of sub-query result, leading to duplicate evaluation

Other approaches to dealing with such problems exist. Cooperative file scan among multiple queries [2,7] focuses on attaching queries started later to already active scans. However, as indicated in [7], such a strategy is not effective when queries scan different ranges rather than the full content of a table. Common Subquery Optimization [6] focuses on sharing query plans rather than evaluation results. In contrast to the general business process management, our work is characterized by in-DB SQL dataflow process management.

In this work, we introduce "functional forms" to specify multiple cooperative queries with user-defined functions using a single declarative expression. We propose to (a) extend SQL to express general graph structured dataflow processes beyond query trees; and (b) extend query engine to orchestrate multiple dataflow processes for sharing database retrieval and query evaluation results, without repeated data retrieval, duplicated computation and unnecessary data copying.

Viewing queries as functions applied to relation objects, we introduce a set of meta-operators called Functional Forms (FFs) for constructing new functions from existing ones. We then provide an algebraic system made of queries, user-defined Relation Valued Functions (RVFs)[4], constructive primitive functions and FFs, referred to as the FF-SQL system. Applying a FF to query functions denotes a combined function, and applying that function to relations denotes a FF-SQL query. While a regular SQL query represents tree-structured dataflow, a FF-SQL query represents general graph-structured dataflow. We use FF-SQL to specify a dataflow process made of multiple correlated queries, which may invoke RVFs, for a particular application goal. We also use FF-SQL to specify a group of correlated dataflow processes with common data sources and data dependencies among their operations.

We have extended the PostgreSQL query engine to execute FF-SQL dataflow processes cooperatively without duplicate data access and query evaluation. A specific in-DB cooperative layer is provided to control application dataflows, schedule queries, and interact with query processing. It isolates much of the complexity of data streaming into a well-understood system abstraction. Our implementation will be reported separately.

The rest of this paper is organized as follows: Section 2 introduces the FF-SQL framework; Section 3 shows the FF-SQL specification of cooperative dataflow processes; Section 4 concludes the paper.

2 FF-SQL – An Algebraic Framework for Queries

A FF-SQL system is used to combine queries and user defined RVFs for representing application dataflow graphs. In the following we describe the operators – functional forms, and operands – query functions and RVFs, of a FF-SQL system.

2.1 Relation Valued Functions

As mentioned above, to extend the action capability of the database engine, we have generalized the table valued functions with scalar input to RVFs with relation input [3,4]. In this way RVFs can be treated as a relational operator or data source and integrated to SQL queries.

2.2 Query Variables vs. Query Functions

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

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

A query variable can be viewed as a relation data object, say v_Q, denoting the query result.

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

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

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

```
fQ : < Sales, Customers> → vQ          (bind Sales to $1 and Customers to $2)
```

The major constraint of query functions is *schema-preserving*, 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.

2.3 Functional Forms

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.

2.4 FF-SQL Framework

A FF-SQL 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 FF-SQL system are of one type: they map relations into relations; and they are the operands of FFs. In general, a FF-SQL system comprises the following.

- A set O of objects. An object is either a relation, a query variable, a sequence $<r_1, \ldots, r_n>$ whose elements r_i are objects, or Δ (undefined), T (true), F (false), $\emptyset$ (empty).
- A set F of functions which are query functions, RVFs, construct primitives and their combinations that map objects into objects. Queries (and RVFs) are schema-aware.
- A meta-operator, apply; applying a function f to an object r is expressed as $f:r$, and in case the object is a sequence $<r_1, \ldots, r_n>$, is expressed by $f : <r_1, \ldots, r_n>$;
- A set C of functional forms for combining existing functions to new functions in F;
- A set D of definitions that define some functions in F and assign a name to each.

A set of constructive-primitives are provided but we only list the ones used in this report, such as

- **Selector** $\$i : <r_1, \ldots, r_n> = r_i$
- **Identity** $id : r = r$
- **Constant** !y : x = y (can also be treated as a FF but we opt to treat it as a function)
- **Union** union: $< r_1, r_2 > = r_1 \cup r_2$

A FF is an expression denoting a function. A FF is primarily used to combine queries with RVFs into higher level ones for expressing dataflow graphs, in a style not expressible by the conventional SQL. A subset of FF primitives used later in this paper is listed below (schema preserving is implied):

- **Composition** $(f \bullet g) : r = f(g:r)$
- **Construction** $[f_1 \ldots f_n] : r = < f_1:r \ldots f_n:r >$
- **Condition** $(p \rightarrow f, g) : r = ((p:r)=T \rightarrow f:r; g:r$
- **Map** (Apply to all) $\alpha f : <r_1 \ldots r_n> = < f:r_1 \ldots f:r_n>$
- **Reduce** $/f : r = r==<r_1> \rightarrow r_1;$
  $r==<r_1 \ldots r_n> \ \&\ n>=2 \rightarrow f:<r_1, /f: <r_2 \ldots r_n>>$

Map represents data-parallelism, construction represents task-parallelism. Reduce can be generally used for stepwise merge and aggregate purposes, such as

$/\$2 : <r_1 \ldots r_n> = r_n$ (last)

$/\text{union} : <r_1 \ldots r_n> = r_1 \cup r_2 \ \ldots \ \cup r_n$

To be specific to relational data manipulation, in the FF-SQL system we do not introduce computation primitives other than queries and user-defined RVFs; and we do not introduce constant values such as a number or a string. To have a non-relation constant passed in an RVF, the "constant" primitive can be used. For instance, given the RVF g_{rvf} with argument list (R_1, R_2, k) where k is an integer and R_1, R_2 are relations, then, for applying g_{rvf} to a sequence of relations $<R_1, R_2>$, a composite function G can be defined by (denoted as :=)

$G := g_{rvf} \bullet [\$1, \$2, !k]$

Applying G to $<R_1, R_2>$ can be expressed as

$G : <R_1, R_2> = g_{rvf} \bullet [\$1, \$2, !k] : <R_1, R_2> = g_{rvf} : <R_1, R_2, k> = g_{rvf}(R_1, R_2, k)$

In the FF-SQL system, functions can be defined level by level from query functions/RVFs in terms of FFs. Apply a FF to functions denotes a new function; and apply that function to relations denotes a *FF-SQL query*. A FF-SQL query has the expressive power for specifying a dataflow graph, in the way not possible by a regular query. In the other words, a regular SQL query represents tree-structured dataflows, a FF-SQL query can further represent graph structured dataflows.

The notion of FF is analogous to the function combining form found in FP system [2], however, the FF-SQL system is a declarative system rather than a functional programming system. Besides queries and RVFs, we do not introduce any computational primitives (such as +, - *, /); instead, we only introduce several constructive primitive functions. Note that FFs are also constructive rather than computational. FF-SQL further differs from FP in taking strongly typed (i.e. schema-aware) relations as data objects, where a query is viewed as a relation data consumer and producer, i.e. a data source.

3 FF-SQL Specification of Cooperative Dataflow Processes

In this section we shall illustrate how to use the proposed FF-SQL to express some typical dataflow schemes in the way not expressible by using regular SQL queries.

As shown in Fig 2, the modulated version of Fig. 1, we assume a table, *Sales*, holding shopping transaction data is retrieved, filtered and aggregated by a star-join query Q_1, resulting a relation, *Txs* with the following schema.

[TxDetailID, Customer, Item, DayOfWeek, Amount, Subtotal]

The attribute *Location* is left for further dimensioning the analysis by state, which is not referred to here for simplicity.

In addition, customer shopping behavior is described by a "feature vector" expressing the average daily spending in a week. A feature vector has 7 values (since a week has 7 days) thus can be viewed as a 7 dimension point in the feature space. For customer segmentation, they are clustered based on such feature vectors representing one aspect of their shopping behaviors. We also assume the existence of initial clusters, each identified by a *cid* and having a centroid feature vector, *cv*.

3.1 The High-Level Function

The star-join result of query Q_1 is delivered to RVF *CF* for generating collaborative filtering matrix, RVF *AR* for generating association rules, and a composite function *CL* for clustering the customers based on their shopping behavior feature vectors. The feature vectors are extracted from the result of Q_1 by query functions Q_f followed by RVF "*extract*". The initial and updated clusters are kept in relation Centroids [*cid*, *cv*].

[Query Variables]

Q_1 = SELECT txDetail-id, item, customer, dayOfWeek, amount, subtotal
FROM Sales, Customers, Product, Time WHERE /* star-join conditions */;

Q_2 = SELECT cid, cv FROM Centroids;

[RVFs]

$AR(Q_1)$	- mining cross-selling association rules
$CF(Q_1)$	- collaborative filtering of customer shopping preference

[Composite Function]

$CL(Q_1, Q_2)$ - cluster customers based on the "feature vectors" representing their average daily spending in a week

Then multiple cooperative dataflow processes are expressed by the following single FF-SQL function.

[FF-SQL main function]

$[AR \bullet \$1, CF \bullet \$1, CL] : <Q_1, Q_2>$

Applying it to data objects $<Q_1, Q_2>$ forms the following FF-SQL query

$[AR \bullet \$1, CF \bullet \$1, CL] : <Q_1, Q_2>$

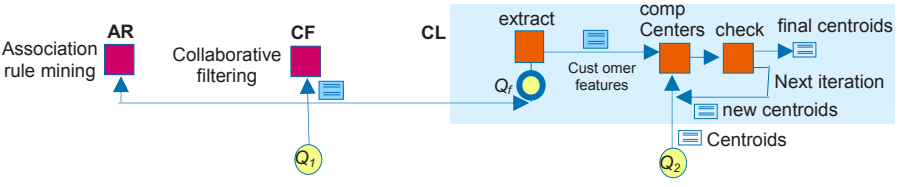

Fig. 2. Cooperative analytics applications with query result sharing

3.2 The Cluster Function

Now let us refine the clustering function CL. It is based on the *k*-means 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 centroids of natural clusters in the data. It assumes that the object attributes form a vector space. The objective it tries to achieve is to minimize total intra-cluster variance. In our example, customers are clustered by the average daily spending in a week which is represented as a 7-valued vector corresponding to the 7 days in a week, or a 7 dimension point in the feature space.

In this K-Means clustering example, the customers are kept in a derived relation

CustomerFeatures [*customer*, *pv*]

where *pv* stands for the feature vector of a customer. The clusters are stored in relation

Centroids [*cid*, *cv*])

where *cid* stands for the ID of a given centroid, and *cv* stands for its feature vector.

For simplicity, let us abbreviate relation CustomerFeatures by P (since a vector can be considered as a 7-dimension Point), and Centroids by C.

[Query Function]

Q_f = SELECT customer, dayOfWeek, SUM(subtotal) AS spending FROM $1
GROUP BY customer, dayOfWeek;

[RVFs]

The RVF

$extract(Q_f(Q_1))$

is used to build feature vectors; it returns relation P (CustomerFeatures). The RVF

$compCenters$: <P, C> → C'

is used to derive a new set of centroids, i.e. a new instance of relation C, from relations P and C in a single iteration; which has the following two steps:

- the first step is for each customer in relation P to compute its distances to all centroids in relation C and assign its membership to the closest one, resulting an intermediate relation Nearest_centroids [pv, cid];
- the second step is to re-compute the set of new centroids based on the average location of member vector points.

After each iteration, the newly derived centroids, C', are compared to the old ones, C, by another RVF

$check$: <C', C> → {T; F}

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

Our goal is to define a FF-SQL query

CL : <P, C>

that derives, in multiple iterations, the centroids of the clusters with minimal total intra-cluster variance, from the initial C relations towards the final instance of relation C.

During the *CL* computation, the relation C is updated in each iteration, but the relation P remains the same. A key requirement is to avoid repeated retrieval/derivation of either relation C or relation P from the database, which should be explicitly expressible at the language presentation level.

The function *CL* is defined by the following.

$comp$:= [$1, $2, $compCenters$];

$renew$:= ($check$ • [$2, $3] → $3; $renew$ • $comp$ • [$1, $3]) ;

CL := $renew$ • $comp$ • [$extract$ • Q_f • $1, $2];

Applying function *CL* to the points, P, and the initial centroids, C, for generating the converged centroids is expressed by the FF-SQL query

CL : <P, C>

The execution of FF-SQL query *CL* : <P, C> is explained as below.

- *comp*, i.e. [$1, $2, *compCenters*], maps relations P and C to a list of relations P, C and C'.

  [$1, $2, $compCenters$] : <P, C> → <P, C, C'>

  where C' is derived by *compCenters* : <P, C>

- then <P, C, C'> becomes the input of *renew*, where *check* • [$2, $3] is applied to C and C', i.e.

  $check$ • [$2, $3] : <P, C, C'> = $check$: <C, C'> → {T; F}

If T is returned the *CL* function terminates with C' (the 3rd element in the above sequence) as its result; otherwise goes to the next iteration;

else if the above check fails, *renew* • *comp* • [$1, $3] is applied for the next iteration for re-generating a set of centroids, say C", as

renew • *comp* • [$1, $3] : <P, C, C'> = *renew* • *comp* : <P, C'>
= *renew* • [$1, $2, *compCenters*] : <P, C'> = *renew* : <P, C', C">

With the above CL definition, the refined main function for these three cooperative applications is specified in the following way.

[FF-SQL main function]

main := [*AR* • $1, *CF* • $1, *CL*]
:= [*AR* • $1, *CF* • $1, *renew* • *comp* • [*extract* • Q_f • $1, $2]]

comp := [$1, $2, *compCenters*];

renew := (*check* • [$2, $3] → $3; *renew* • *comp* • [$1, $3]) ;

The FF-SQL query for applying *main* to the data objects, i.e. the results returned from queries Q_1, Q_2, is expressed by the following.

main : <Q_1, Q_2> = [*AR* • $1, *CF* • $1, *CL*] : <$Q_1$, Q_2> = <*AR*:Q_1, *CF*:Q_1, *CL*:< Q_1, Q_2>>

As described later, the different versions of the instances of relation C output from *compCenters* in multiple iterations, actually occupy the same memory space.

4 Conclusions

In this work we proposed an approach for pushing data-intensive analytics down to database engines for high-performance and secured execution: integrating general analytic operations into SQL queries, handling general graph structured dataflows, and executing multiple dataflow processes with common subqueries or data sources.

To allow general analytic operations to be naturally and efficiently integrated with SQL queries, we support *RVF* at SQL language level. To *declaratively* express graph based dataflow we extend SQL to FF-SQL. To execute a group of correlated dataflow processes cooperatively without repeated data retrieval, duplicated computation and unnecessary buffering, we extend the query engine with a memory-based, embedded middleware layer.

The major advantage of FF-SQL lies in its expressive power for specifying complex dataflows. Specifically, a group of correlated SQL dataflow processes with common data sources or subqueries can be specified by a single, integrated FF-SQL dataflow definition, which provides the basis for their cooperation inside the database engine. The advantage of FF-SQL also lies 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.

In this paper we described FF-SQL informally; the detailed formalisms, including the algebraic laws on FFs, will be documented separately. We are also developing support for FF-SQL by building a middleware layer inside the PostgreSQL database engine that deals with data buffering, dataflows, control flows and function scheduling.

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. Cao, Y., Das, G.C., Chan, C.-Y., Tan, K.-L.: Optimizing Complex Queries with Multiple Relation Instances. In: ACM SIGMOD 2008 (2008)
3. Chen, Q., Hsu, M.: Data-Continuous SQL Process Model. In: Proc. 16th International Conference on Cooperative Information Systems, CoopIS 2008 (2008)
4. Chen, Q., Hsu, M., Liu, R.: Extend UDF Technology for Integrated Analytics. In: Proc. 10th Int. Conf. on Data Warehousing and Knowledge Discovery, DaWaK 2009 (2009)
5. 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)
6. Tao, Y., Zhu, Q., Zuzarte, C.: Exploring Common Subqueries for Complex Query Optimization. In: IBM Centre for Advanced Studies Conference (2002)
7. Zukowski, M., Héman, S., Nes, N., Boncz, P.: Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In: VLDB (2007)