DEIM Forum 2012 D2-5

# SQLET: A Database Programming Language and Execution Environment for Parallel SQL Processing running on Plain RDBMSs

Makoto YUI† and Isao KOJIMA†

† Information Technology Research Institute, National Institute of Advanced Industrial Science and Technology, Japan

1–1–1 Umezono, Tsukuba, Ibaraki 305–8568 Japan

**Abstract** This paper introduces a new database programming language and execution environment, named SQLET, for analytical SQL processing on shared-nothing RDBMSs. The goal of our system is making parallel SQL processing system over bunch of plain RDBMSs as well as providing a domain-specific language giving the user a powerful mechanism to facilitate bunch of nodes and processors. We discuss the issues and design objectives, as well as the unique requirements and challenges that we found through building SQLET.

**Key words** Parallel Database, Database Programming Language, DSL, Horizontal Partitioning

## 1 Introduction

Many successful online service providers (e.g., facebook, twitter and mixi) use horizontal table partitioning to cope with enormous growths in transaction volume and size of application databases. Each of them typically builds a custom infrastructure of so-called *database sharding*, an application-level range (or hash) partitioning technique on shared-nothing servers, particularly due to lack of a corresponding feature for scaling on open source databases. These application-level load balancing schemes work well for OLTP uses (e.g., look up by user-id), but not for analytical uses such as daily statistical reporting over the partitioned databases.

Fig. 1 illustrates a typical configuration of database sharding. In this example, there are one or more replicas for data durability and shared 1 to 3 are used for managing a user table of the ID range 1 to 15,000. Each database in a shared manages 5,000 rows until 2009–2011; however, shared 4 that manages a user table of the ID range 15,000 to 30,000 is added in 2012. It is notable that hardware configuration, size of each partition and the number of replicas could vary over time, e.g., between 2012 and the before. The newer servers that have larger memory spaces and/or cutting-edge processors get a greater share of the workload than others by design. Though the physical database design might work well at the beginning, it is mandatory that workloads change over time and then dynamic load balancing is required. Dynamic load balancing becomes feasible by dynamically redistributing data.

We assume the above setting given throughout this paper and then introduce a new database programming language and execution environment dedicated to the situation. The goal of this paper is making parallel SQL processing system over bunch of plain RDBMSs and giving the users a powerful mechanism to facilitate bunch of nodes and processors. This gives unique requirements and challenges for middleware-based parallel query processing to parallel databases as follows:

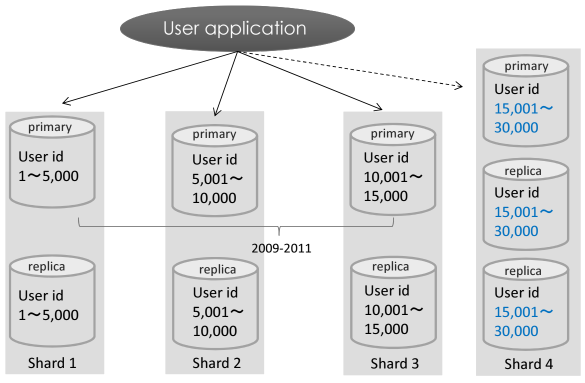

Fig. 1 Range partitioning by user ID

1) *Partitioning information is managed outside databases.* So, in contrast to parallel databases, the parallel query processing module has to use partitioning information in various external modules. A standardized way to load partitioning information is expected for the execution environment.

2) *Support for online reprovisioning.* A database administrator (DBA) may initially allocate 10 nodes to accomplish warehouse processing. The load later rises, and the desire is to allocate 20 nodes to the task originally done by 10. This requires the database to be repartitioned over double the number of nodes. Hardly anybody wants to take the re-

quired amount of down time to dump and reload the DBMS. A much better solution is for the DBMS to support online reprovisioning, without going offline.

3) *Avoid impacts on online databases.* Being linked to the above issue, the execution environment must avoid considerable impacts on the online databases and online services. Though requirements from business world are becoming more real-time, not every client can justify the costs of real-time data warehousing to online transactions.

With these issues in mind, we introduce a new database programming language and execution environment, named *SQLET*, for analytical SQL processing on shared-nothing RDBMSs. We assume SQLET as a building block of a parallel database environment and made a design choice that advanced features such as online reprovisioning are going to be built on the top of it. The objective of this paper is not only introducing a technique for parallel SQL processing but also proposing an software architecture enabling that transactional and analytical processing reside in one single system. Though the development is partly on-going, we discuss the issues and design objectives, as well as the unique requirements and challenges that we found in building parallel analytical SQL processing on top of shared-nothing RDBMSs.

The rest of this paper is organized as follows. Section **2** introduces the design and implementation of SQLET language and its execution environment. Section **3** discusses characteristics that a map-reduce style execution essentially involves. The following Section **4** introduces the advanced use of SQLET. We conclude the paper in Section **5**.

## 2 Design and Implementation

### 2. 1 Analytical SQL Processing on Slave Databases

Both transactional databases and analytical databases are based on the same database theory but analytical operations are usually moved off from operational databases to a dedicated data warehouse. This design imposes the additional management of extracting, transforming and loading data, as well as controlling the redundancy. Though, for many years, the discussion seemed to be closed and enterprise data was split into OLTP databases and data warehouses, recent demands for timely and fast decision making opened an opportunity to reconsider the common practice [1, 2].

We take the same position to them but take a different approach to those who uses a columnar database [1, 2] or main memory databases [2] towards the achievement; we use slave databases (that has *k-safety* [3]) in database shards, as one in Fig. 1, to avoid the impacts on online databases. Those slave databases are typically configured to replicate the master databases asynchronously and are used only for data redundancy and high availability. We propose to use them for analytical purposes as well as availability.

### 2. 2 SQLET Execution Environment

Following on the successful commercial systems (say SQL/MapReduce) [4–6], we introduce MapReduce [7] functionality into relational databases. We introduce a new database programming language and execution environment that has a SQL/MapReduce functionality, named SQLET (pronounced *secret*), for analytical SQL processing on shared-nothing RDBMSs.

**Differences to Other SQL/MapReduce**

The major differences between existing SQL/MapReduce systems and SQLET are:

- First, SQLET is designed to run on any relational databases. It also support mixed environments of different databases. Columnar databases can be used only for *reduce* processing when the table is partitioned over row-store databases. This gives us an optimization opportunity to use the right database in the right job.
- Second, SQLET does not control partitioning. Instead, the SQLET execution environment provides a feature to load partitioning information, described in a general CSV format, from external modules. We consider this design is particularly beneficial where the database is already partitioned as in Fig. 1.
- Third, SQLET is designed to provide building blocks for making parallel database environments. It is usual that parallel constructs of parallel databases (e.g., *shuffle* operator in algebra [8]) are hidden from the user who operates higher-level concepts of relational database; primitive operators are proprietary assets of each particular database product and are never disclosed. Our motivation here is, on the contrary, giving the users a flexibility to use the parallel constructs such as shuffling because we believe that it helps users to build their requisites such as online reprovisioning. Of course, users can restrict the flexibilities at their higher-level abstraction.

**Architecture of SQLET**

Fig. 2 shows the architecture of the SQLET execution environment. SQLET takes a master-less/multi-master architecture. A processor element (PE) that the client connected will be the master in the session. The APIs provided for the SQLET is through Thrift RPC [9], a cross-language RPC service. Users invoke parallel SQL processing through the Thrift and retrieve the final output table through standard database access APIs such as JDBC and ODBC. There are two essential service endpoints in SQLET:

- *executeCommand* that takes a parsed SQLET query (we call it SQLET command) and returns the result.
- *executeQueries* that takes sequence of SQLET queries and executes them in a single session.

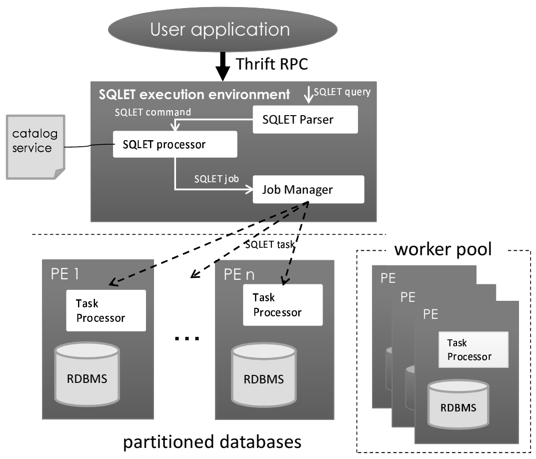

Fig. 2 Architecture of the SQLET execution environment

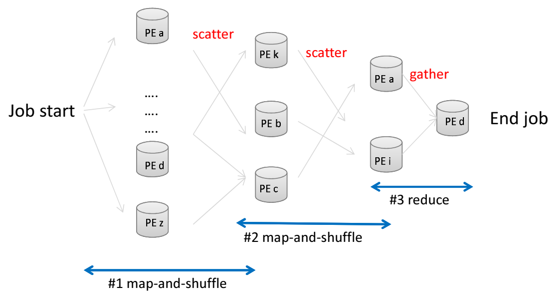

Fig. 3 An example of MapReduce processing using SQLET

```
map {
  alter table name set c1 = "test1";
  select * from table where ..;
} catalog_name = 'tpch100';
```

The SQLET parser transforms SQLET queries into a sequence of SQL commands and the SQLET processor evaluates the sequence of SQLET commands. Taking the sequence of SQLET commands and catalog information as the input, the SQLET processor submits SQLET jobs to the job manager. Each SQLET job is responsible for submitting queries as SQLET tasks to the right processor element.

The *map* queries of SQLET run on each partitioned database where data exists in parallel. Each processor element in the worker pool are used for aggregate processing in the *reduce* phase. The catalog service takes a key role in controlling the *map* and *reduce* query execution where they are actually executed; it manages the following two information: *partitions* and *reducers*. The *partitions* involves processor elements where some partition is stored as well as a master/slave attribute. The *map* queries are launched at the every processor element listed in *partitions*. The *reducers* involves processor elements where the map output are going be stored (we call them reducer). To control the *partitions* and *reducers*, the following DDLs are prepared that replaces catalog values.

```
LOAD PARTITIONS INTO CATALOG <name> FROM FILE <uri>;
LOAD REDUCERS INTO CATALOG <name> FROM FILE <uri>;
```

Various standard protocols (e.g., http protocol) are supported for the URI format to load the CSV data into the execution environment.

## 2.3 SQLET Language

SQLET is a domain-specific language for parallel analytical SQL processing on shared-nothing RDBMSs. There are mainly three types of SQLET commands that form the essence of SQLET language: *map*, *map-and-shuffle* and *reduce*. We explain these directives by giving examples.

The *map* is the most simple directive that takes any SQL queries and just forward the queries to the processor elements specified in the given catalog. Any possible SQL queries are accepted in the block statement. The execution does not have the return value. The *map* directive is useful for a batch operation as well as DDL operations over the specified databases. The default catalog is used when *catalog_name* is not specified.

```
map-and-shuffle {
  SELECT .. FROM ..;
} distributed by hash(c1, c2), catalog_name='tpch100';
```

The *map-and-shuffle* invokes a SELECT statement in the partitioned databases and distributes the execution result among reducers. A single SELECT statement is valid for the statement of *map-and-shuffle* directive. The results of SELECT statement are exported into a file using a COPY INTO on each execution. And then, the result is distributed over the specified reducers. The *distributed by* clause controls how to distribute the result tuples in the *shuffle* phase. In the following example, the first and the second column are used for hash partitioning. Range and round-robin partitioning is also to be supported. As seen in Fig. 3, the reducers used in the first *map-and-shuffle* execution could be the inputs of the next *map-and-shuffle* or *reduce*. If there are more than one replica, the *map-and-shuffle* execution supports a task-level failure handling on node failures and a speculative execution to cope with straggler nodes as those seen in the original MapReduce [7].

```
reduce at <PE> {
  SELECT .. FROM ..;
} catalog_name='catalogName';
```

The *reduce* directive is used for the final aggregation as seen in Fig. 3. It invokes a SELECT statement in the partitioned databases and distributes the results as similar to *map-and-shuffle*. The *reduce* directive can be considered as a syntax sugar of *map-and-shuffle* with only one reducer. The distinction is that *reduce* is used for gathering the map output while *map-and-shuffle* is for scattering the map output. The gathering destination is specified through "at <PE>" clauses. In contrast to *map-and-shuffle*, *reduce* needs not to parse the map output for shuffling; it just forward the result to the gathering destination.

## 3 Performance Model

This section describes a performance model of SQLET to highlight what we found through building SQLET. We introduce the computation complexbility of MapReduce to see characteristics that map-reduce style execution essentially involves.

### 3.1 Computational Cost of MapReduce

MapReduce is inspired by the map and reduce functions commonly used in functional programming, although MapReduce frameworks [7,10] are not making full use of the characteristics behind thier original forms [11].

The original forms of *map* and *reduce* can be formalized as follows:

- **map** $f[x_1, x_2, \cdots, x_n] = [fx_1, fx_2, \cdots, fx_n]$
  applies the function $f$ to each element of the list. This can be computed at the same time to $f$ when assumming enough processors for the parallel computation.
- **reduce** $(\oplus)\ [x_1, x_2, \cdots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$
  computes the reduction of the list as with the associative binary operator $\oplus$. $\oplus$ can be computed using $O(logN)$ parallel time ($N$ denotes the size of the list) on a tree-like structure when $\oplus$ is associative. If the binary operation is also commutative, then the order of combining results from sub-reductions can be arbitrary.

It is known that MapReduce has bottlenecks in reduce phase where single or few reducers are mandatory as shown in the following aggregate query.

```
Q1:  SELECT SUM(CountWords(txtField))/COUNT(*) FROM T1
```

Google's Dremel [12] showed significant (order of magnitude) advantages over MapReduce (even with a columnar storage) for this sort of aggregate intensive computation because of its efficienties in partial aggregations [13].

**Analysis and Reduction of Glitches in MapReduce**

Two reasons can be considered why Dremel [12] showed the order of magnitude advantage over MapReduce.

(1) *Optimization for large aggregations.* Dremel hierarchically executes aggregate queries on a multi-level serving tree [12]. With large aggregations, having more levels in the serving tree can result in speedups since there is more partial aggregation of intermediate results. Dremel can compute aggregation queries like Q1 at $O(log(n))$ parallel time whereas MapReduce requires $O(n)$ parallel time for computing them.

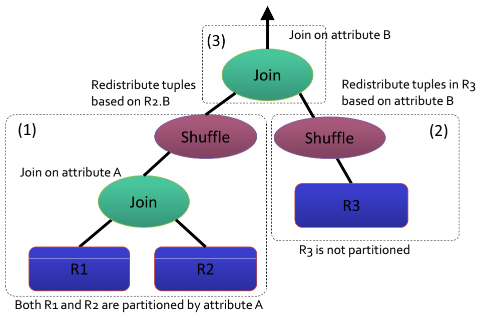
(3)
Join on attribute B
Join
Redistribute tuples based on R2.B
Redistribute tuples in R3 based on attribute B
(1)
Shuffle
Shuffle
(2)
Join on attribute A
Join
R3
R3 is not partitioned
R1
R2
Both R1 and R2 are partitioned by attribute A

Fig. 4 An example that requires cross-shared joins

(2) *Avoidance of intermediate results materialization on disks.* MapReduce materializes intermediate results at several points in time to checkpoint the computing progress done by mappers and reducers. First, the map output is sorted and stored in local disks in order to deal with node failures. Second, the map outputs are copied (pulled) to the reducer's memory if they are small enough; otherwise, they are copied to disk. As the copies accumulate on reducers, a k-way (external) merge sort is carried to merge them on each reducer. Moreover, both mapper and reducer rely on a distributed file system as the underlying storage layer to read input and store output. The output of reducers are stored and replicated in a distributed file system for each iteration and the output becomes the input to the mappers. This requires lots of I/Os and unnecessary computations and makes MapReduce unfeasible for iterative algorithms such as partial aggregation. On the other hand, Dremel performs partial aggregation using a serving tree so that the intermediate results as the data is streamed from the leaf nodes up to the root data, not through disks but through the network.

We found, in the previous work, that queries like TPC-H Q10 [14] becomes a bottleneck where heavy computation on the relatively large intermediate results remains in reduce phase [15]. The efficiency in reduce is a vital developmental requirement for SQLET and SQLET exploits the $O(logN)$ execution for reduce processing as shown in Fig. 3. Google MapReduce [7] and its open source implementation Hadoop [10] does not support a divide-and-conquer execution of reduce. Consequently, the efficiency in reduce is a possible advantage of SQLET to MapReduce.

## 4 Advanced Use of SQLET

SQLET provides users building blocks to build higher-level concepts. This section shows how users can use SQLET as building blocks by giving practical examples.

### 4.1 Cross-Shard Joins

It is known that parallel databases are efficient particularly when the table partitioning is a priori arranged for the target workloads so that on-the-fly data shuffling [8]

```
let $p1 := spawn map-and-shuffle { (1)
  select r1.b as b from R1 join R2 on R1.a = R2.B;
} distributed by hash(c1), catalog_name='test', output_table='leftt';
let $p2 := spawn map-and-shuffle { (2)
  select b from R3;
} distributed by hash(c1), catalog_name='test', output_table='rightt';
sync $p1, $p2; – wait for the processes finish executing
– override the partitions and reduces information for reduce.
load partitions into catalog 'test' from file "/tmp/reducer1.csv";
load reducers into catalog 'test' from file "/tmp/reducer2.csv";
reduce at 'PE4' { (3)
  select * from leftt l join rightt r on l.b = r.b;
} catalog_name='test';
```

Fig. 5 A SQLET job to process queries as shown in Fig. 4

can be avoided (or reduced). However, on-the-fly shuffling is mandatory when multi-way joins are carried. Assuming queries of the form $Q = \{target|qualification\}$ where target is a list of projected attributes and qualification is a list of equi-joined attributes, let a multi-join query be $Q = \{R_1.A, R_2B|R_1.A = R_2.A \wedge R_2.B = R_3.B\}$. Then not all of the three relations can be partitioned since the first join predicate requires $R_2$ be partitioned by attribute $A$ and the second join predicate requires $R_2$ be partitioned by attribute $B$. A certain conflict exists between these requirements [16].

This section shows the way to process such multi-way joins. Fig. 4 is an example of multi-way joins; table R1 and R2 are joined on attribute A and consequently a join operation is performed with table R3 using attribute B. R1 and R2 are adequately partitioned using attribute A but R3 is not partitioned yet there. Then, the join operation between table R1 and R2 can be performed in parallel. However, two shuffle operations are required to process the consequent join (in parallel). To process this algebra, *map-and-shuffle* commands are used at (1) and (2) respectively and *reduce* command is carried at (3) as shown in Fig. 5. Note here that (1) and (2) should be run in parallel.

**Parallelization Dialect**

To cope with the above issue, we introduce *spawn* and *sync*, parallelization controls derived from Cilk [17] as seen in Fig. 5. The *spawn* invokes a SQLET command in a child thread and *sync* waits for the specified processes to finish. The *spawn* takes not only single SQLET statement but also multiple statements. Nested execution of the parallelization dialects is also supported.

## 5 Conclusions

In this paper, we introduced a new database programming language and execution environment, named SQLET, for analytical SQL processing on shared-nothing RDBMSs. We discussed the issues and design objectives, as well as the unique requirements and challenges that we found in building parallel analytical SQL processing on top of shared-nothing RDBMSs.

A major on-going issue of SQLET is how to bring *pipeline parallelism* into the language design. It would be interesting to introduce pipeline parallelism into the interactive SQLET execution as in Fig. 3. We are exploring to use a streaming database system [18] for the intermediate stages for the purpose.

## Acknowledgment

This work was partly supported by JSPS grant-in-aid for research activity start-up #22800086.

**REFERENCES**

[1] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proc. SIGMOD*, pages 61–74, 2009.
[2] Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. A Hybrid Row-Column OLTP Database Architecture for Operational Reporting. In *Proc. BIRTE*, 2008.
[3] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In *Proc. VLDB*, pages 553–564, 2005.
[4] Eric Friedman, Peter M. Pawlowski, and John Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.
[5] Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Erik Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proc. SIGMOD*, pages 1165–1176, 2011.
[6] EMC Corporation. Greenplum. http://www.greenplum.com/.
[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, pages 137–150, 2004.
[8] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: Friends or foes? *Commun. ACM*, 53(1):64–71, January 2010.
[9] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.
[10] Hadoop. http://hadoop.apache.org/.
[11] Ralf Lämmel. Google's MapReduce programming model – Revisited. *Science of Computer Programming*, 70(1):1–30, January 2008.
[12] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1):330–339, 2010.
[13] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260, 2009.
[14] T.P.P. Council. TPC-H Benchmark Specification. http://www.tpc.org/tpch/.
[15] Makoto Yui and Isao Kojima. A Parallel Database Architecture Avoiding Tuple Redistribution. *IPSJ TOD*, 4(4):11–33, Dec 2011.
[16] C. Liu and H. Chen. A hash partition strategy for distributed query processing. In *Proc. EDBT*, pages 371–387,

1996.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. PPOPP*, pages 207–216, 1995.

[18] EsperTech Inc. Esper. http://esper.codehaus.org/.