Scalable Queries over Log Database Collections

Minpeng Zhu, Khalid Mahmood, and Tore Risch

Department of Information Technology, Uppsala University, Uppsala 75237, Sweden
{minpeng.zhu, khalid.mahmood, tore.risch}@it.uu.se

Abstract. Various business application scenarios need to analyse the working status of products, e.g. to discover abnormal machine behaviours from logged sensor readings. The geographic locations of machines are often widely distributed and have measurements of logged sensor readings stored locally in autonomous relational databases, here called *log databases*, where they can be analysed through queries. A global meta-database is required to describe machines, sensors, measurements, etc. Queries to the log databases can be expressed in terms of these meta-data. FLOQ (Fused LOg database Query processor) enables queries searching collections of distributed log databases combined through a common meta-database. To speed up queries combining meta-data with distributed logged sensor readings, sub-queries to the log databases should be run in parallel. We propose two new strategies using standard database APIs to join meta-data with data retrieved from distributed autonomous log databases. The performance of the strategies is empirically compared with a state-of-the-art previous strategy to join autonomous databases. A cost model is used to predict the efficiency of each strategy and guide the experiments. We show that the proposed strategies substantially improve the query performance when the size of selected meta-data or the number of log databases are increased.

1 Introduction

Various business applications need to observe the working status of products in order to analyse their proper behaviours. Our application is from a real-world scenario [11], where machines such as trucks, pumps, kilns, etc. are widely distributed at different geographic locations and where sensors on machines produce large volumes of data. The data describes time stamped sensor readings of machine components (e.g. oil temperature and pressure) and can be used to analyse abnormal behaviours of the equipment. In order to analyse passed behaviour of monitored equipment, the sensor readings can be stored in relational databases and analysed with SQL. In our application area, data is produced and maintained locally at many different sites in autonomous relational DBMSs called *log databases*. New sites and log databases are dynamically added and removed from the federation. The number of sites is potentially large, so it is important that the query processing scales with increasing number of sites. A global meta-database enables a global view of the working status of all machines on different sites. It stores meta-data about machines, sensors, sites, etc.

A particular challenge in our scenario is a scalable way to process queries that join meta-data with data selected from the collection of autonomous log databases using standard DBMS APIs. This paper proposes two strategies to perform such joins, namely *parallel bind-join* (PBJ) and *parallel bulk-load join* (PBLJ). PBJ generalizes the *bind-join* (BJ) [4] operator, which is a state-of-the-art algorithm for joining data from an autonomous external database with a central database. One problem with bind-join in our scenario is that large numbers of SQL queries will be sent to the log databases for execution, one for each parameter combination selected from the meta-database, which is slow. Furthermore, whereas bind-join is well suited for joining data from a single log database with the meta-database, our application scenario requires joining data from many sites.

With both PBJ and PBLJ, streams of selected meta-data variable bindings are distributed to the wrapped log databases and processed there in parallel. After the parallel processing the result streams are merged asynchronously by FLOQ.

- With PBJ the streams of bindings selected from the meta-database are bind-joined in the distributed wrappers with their encapsulated log databases. The bind-joins of different wrapped log databases are executed in parallel.
- With PBLJ the selected bindings are first bulk loaded in parallel into a *binding table* in each log database where a regular join is performed between the loaded bindings and the local measurements.

The strategies are implemented in our prototype system called *FLOQ (Fused LOg database Query processor)*. FLOQ provides general query processing over collections of autonomous relational log databases residing on different sites. The collection of log databases is integrated by FLOQ through a meta-database where properties about data in the log databases are stored. On each site the log database is encapsulated by a *FLOQ wrapper* to pre- and post-process queries.

To investigate our strategies, a cost model is proposed to evaluate the efficiency of each strategy. To evaluate the performance we define fundamental queries for detecting abnormal sensor readings and investigate the impact of our join strategies. A relational DBMS from a major commercial vendor is used for storing the log databases.

In summary the contributions are:

- Two join strategies are proposed and compared: parallel bind-join and parallel bulk-load join, for parallel execution of queries joining meta-data with data from collections of autonomous databases using external DBMS APIs.
- A cost model is proposed to evaluate the strategies.
- The conclusions from the cost model are verified experimentally.

The rest of this paper is organized as follows: Section 2 overviews the FLOQ system architecture and presents the scenario and queries used for the performance evaluation. Section 3 presents the join strategies and the cost model used in the evaluation. Section 4 presents the performance evaluation for the join strategies. Section 5 describes related work. Finally, Section 6 concludes and outlines some future work.

2 FLOQ

Fig. 1 illustrates the FLOQ architecture. To analyse machine behaviours, the user sends queries over the integrated log databases to FLOQ. FLOQ processes a query by first querying the meta-database to find the identifiers of the queried log databases containing the desired data, then in parallel sending distributed queries to the log databases, and finally collecting and merging the distributed query results to obtain the final result. Scalable parallel processing of queries making joins between a meta-database and many large log databases is the subject of this paper.

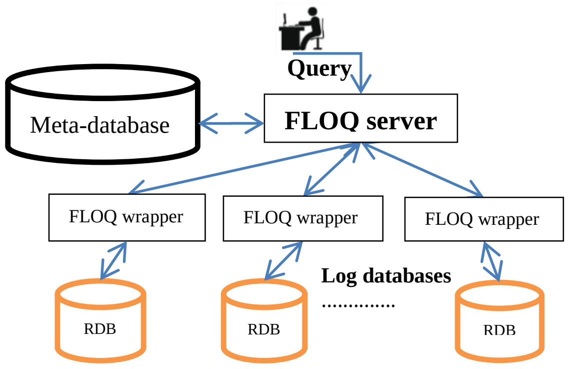


Fig. 1. FLOQ system architecture

Each log database is encapsulated with a *FLOQ wrapper* called from the FLOQ server to process queries over the wrapped log database. A FLOQ wrapper contains a full query processor which enables, e.g. local bind-joins between a stream of bindings selected from the meta-database and the log database. Parallel processing is provided since the FLOQ wrappers work independently of each other. Each FLOQ wrapper sends back to the FLOQ server the result of executing a query as a stream of tuples. The results from many wrappers are asynchronously merged by the FLOQ server while emitting the result to the user. Details of the query processor are described in [10], [13, 14] and are outside the scope of this paper.

2.1 The FLOQ schema

The schema for the FLOQ meta-database is shown in Fig. 2(a). The table *MachineModel(m, mmn, descr, mmanuf)* stores data about machine models, i.e. a unique machine model identifier *m*, along with its name *mmn*, description *descr*, and manufacturer *mmanuf*. The table *MachineInstallation(mi, m, sid)* stores meta-data about each machine installation, i.e. a unique machine installation identifier *mi*, its installed site *sid* and its machine model identifier *m* (foreign key). The table *SensorModel(sm, sname, smanuf)* stores information about sensor models, i.e. a unique sensor model identifier *sm*, the sensor model name *sname*, and its manufacturer *smanuf*. The table *SensorInstallation(si, mi, sm, ev)* stores the sensor installation information, i.e. a sensor installation identifier *si*, the machine installation *mi* of *si*, the sensor model *sm*, and the expected measured value *ev*. The columns *m* and *sid* in table *MachineInstallation* are foreign keys in tables *MachineModel* and *Site*, respectively. The column *mi* in table *SensorInstallation* is foreign key to *MachineInstallation*.

```
MachineModel(m, mmn, descr, mmanuf)
MachineInstallation(mi, m, sid)
SensorModel(sm, sname, smanuf)
SensorInstallation(si, mi, sm, ev)
Site(sid, name, logdb)
```

Fig. 2(a). Meta-database schema

```
Measures(mi,si,bt,et,mv)
```

Fig. 2(b). Log table at each site

```
VMeasures(logdb,mi,si,bt,et,mv)
```

Fig. 2(c). Integrated view in FLOQ server

The table *Site(<u>sid</u>, name, logdb)* stores information about the sites where the log databases are located: a numeric site identifier *sid*, its *name*, and an identifier of its log database, *logdb*. A new log database is registered to FLOQ by inserting a new row in table *Site*. Each site presents to FLOQ its log data as a temporal local relation *Measures(<u>mi, si, bt</u>, et, mv)* (Fig. 2(b)) representing measurements from the sensors installed on the machines at the site, i.e. temporal local-as-view [5] data integration is used. For a machine installation *mi* at a particular site the local view presents the measured readings from sensor installation *si* in the valid time interval *[bt,et)*. The columns *mi* and *si* in *Measures* are foreign keys from the corresponding columns in the meta-database tables *MachineInstallation* and *SensorInstallation*, respectively.

The view *VMeasures* (Fig. 2(c)) in FLOQ integrates the collection of log databases. It is logically a union-all of the local *Measures* views at the different sites. In *VMeasures* the attribute *logdb* identifies the origin of each tuple. Through the meta-database users can make queries over the log databases by joining other meta-data with *VMeasures*. Since the set of log databases is dynamic it is not feasible to define *VMeasures* as a static view; instead FLOQ processes queries to *VMeasures* by dynamically submitting SQL queries to the log databases and collecting the results. In the experiments we populate the meta-database and the log databases with data from a real-world application [11].

2.2 Example Queries

Q1 in Fig. 3 is a simple query that retrieves unexpected sensor readings. It returns machine identifiers *mi* together with the time intervals *[bt,et)* when a sensor on the machine has measured values *mv* higher than the expected values *ev* by a threshold parameter *th* on line 5 marked '?'.

```
Q1:
1  SELECT m.mi, m.bt, m.et
2  FROM Measures m, Site s,
3       MachineInstallation mi,
4       SensorInstallation si
5  WHERE m.mv > si.ev+? AND
6        mi.mi > ? AND
7        si.mi = mi.mi AND
8        m.si = si.si AND
9        m.logdb = s.logdb AND
10       s.sid < ?
```

Fig. 3. Query Q1

```
Q2:
1  SELECT count(*)
2  FROM Measures m, Site s,
3       MachineInstallation mi,
4       SensorInstallation si
5  WHERE m.mv > si.ev+? AND
6        mi.mi > ? AND
7        si.mi = mi.mi AND
8        m.si = si.si AND
9        m.logdb = s.logdb AND
10       s.sid < ?
```

Fig. 4. Query Q2

Query *Q1* is used for the basic scalability experiments. It contains a simple numerical expression over the log database view in terms of *th*. On line 6 there is a constraint on the selected machine identifiers *mi* and on line 10 the selected sites *sid* are restricted. The experiments are scaled by varying these parameters. The number of log databases is varied by restricting *sid*, the amount of data selected from each log database is varied by *th*, and the number of bindings selected from the meta-database is varied by *mi*.

Query *Q2* in Fig. 4 is similar to *Q1*, the difference being that it applies an aggregate function over *Q1*, i.e. it computes the number of faulty sensor readings. Here only a single value is returned from each log database. The purpose of the query is to investigate the join strategies without concerning the overhead of transferring substantial amounts of data back to the client.

```
Q3:
SELECT m.mi, m.bt, m.et
FROM Measures m, Site s,
     MachineInstallation mi,
     SensorInstallation si
WHERE abs(m.mv-si.ev)/si.ev>? AND
      si.mi = mi.mi AND
      m.si = si.si AND
      m.logdb=s.logdb
```

Fig. 5. Query Q3

```
Q4:
SELECT m.mi, m.bt, m.et
FROM Measures m, Site s,
     MachineInstallation mi,
     SensorInstallation si
WHERE si.mi = mi.mi AND
      m.si = si.si AND
      m.logdb = s.logdb AND
      ((m.mv>(1+?)*si.ev and si.ev>0) or
       (m.mv<(1+?)*si.ev and si.ev<0) or
       (m.mv<(1-?)*si.ev and si.ev>0) or
       (m.mv>(1-?)*si.ev and si.ev<0))
```

Fig. 6. Transformed Q3

Query *Q3* in Fig. 5 is an example of a more complex numerical query for identifying machine failures. It detects situations where the relative deviation of sensor readings from *ev* is larger than a threshold parameter we denote *rth*. One property of *Q3* is that the query optimizer of the used DBMS cannot utilize an ordered index on the measured value *mv*, so the entire local table *Measures* on each site will be scanned entirely. This query thus has a high query execution cost for searching the log databases.

Query *Q4* in Fig. 6 is a manually transformed version of *Q3* to expose the index column *mv* of *Measures* table for query optimizer of the DBMS for scalable search. Here all parameter occurrences in the query (marked ?) refer to the supplied value of *rth*. FLOQ automatically makes this algebraic transformation by utilizing the algorithm in [12]. The difference between *Q3* and *Q4* shows the trade-off between full scan and index scan in the log databases enabled by the rewrite. *Q3* is an expensive query compared to *Q4*.

3 Join Strategies

The two strategies, PBJ and PBLJ, for parallel execution of queries joining data between the meta-database and the log databases are illustrated in Fig. 7 and Fig. 8, respectively. With both strategies FLOQ first extracts parameter bindings from the meta-database. The result is a stream of tuples is called the *binding stream B* where each tuple *(i, v_1, v_2, ..., v_p)* is a *parameter binding*. The elements v_1, v_2, ..., v_p of the binding stream are the values of the free variables in the query fragment sent to the log databases. For example, in Q1 the free variables are *(mi, si, ev)*. Each binding tuple is prefixed with a *destination site, i,* identifying where the log database *RDB_i* resides. The parameter binding tuples are joined with measurements in the log databases. Thus the binding stream is split into one site binding stream B_i per log database RDB_i, $B = B_1 \cup B_2 \ldots \cup B_n$, where n is the number of sites. The destination i determines to which site the rest of the tuple, $(v_1, v_2, \ldots, v_p)$, is routed. The join strategies are defined as follows:

PBJ, parallel bind-join: PBJ (Fig. 7) is a generalization of bind-join [4] to handle parallel execution between a common meta-database and a collection of wrapped relational databases RDB_i. On each site i the tuples in the binding stream B_i received by a FLOQ wrapper is bind joined (BJ) with the query σ_i sent to the database RDB_i through parameterized (prepared) JDBC calls. The tuples in the result stream R_i from the JDBC calls are then streamed back to the FLOQ server, where they are merged asynchronously with the result tuples from other sites. With PBJ, a parameterized query is executed many times in each wrapped log database, once for each parameter binding in B_i.

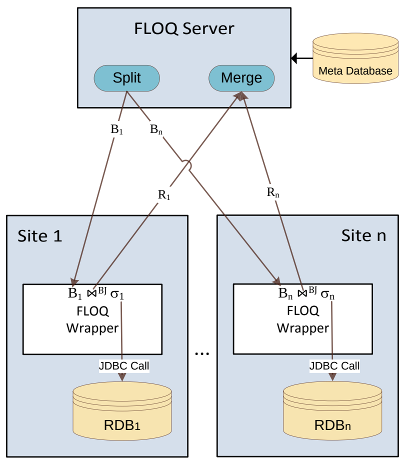


Fig. 7. PBJ

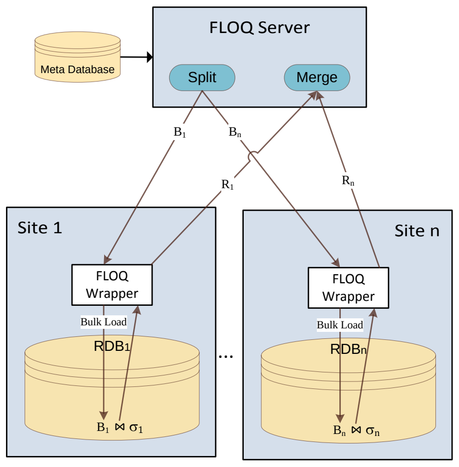


Fig. 8. PBLJ

PBLJ, parallel bulk-load join: With PBLJ (Fig. 8) each FLOQ wrapper first bulk loads the entire binding stream B_i into a *binding table* in RDB_i. When all parameter bindings have been loaded, the system submits a single SQL query to the log database to join the loaded binding table with σ_i. As for PBJ, the result stream R_i is shipped back to the FLOQ server through the wrapper for asynchronous merging. Compared to PBJ, the advantage of this approach is that only one query is sent to each log database. It requires the extra step of bulk loading in parallel the entire parameter streams into each log database, which, however, should be less costly compared to calling many prepared SQL statements through JDBC with PBJ. The bulk loading facility of the DBMS is utilized for high performance.

BJ, regular bind-join: If there is a single log database, PBJ is analogous to BJ and is a baseline in our evaluations. With BJ one prepared SQL query per binding is shipped from the FLOQ wrapper to only one log database, RDB_1.

3.1 Cost Model for Join Strategies

The total cost in terms of response times of the proposed join strategies is divided between the cost of execution in the FLOQ server C_{FLOQ} and the maximum site cost C_i.

$$C_{Join} = C_{FLOQ} + max(\{C_i : i = 1, \ldots, n\}) \quad (1)$$

The total cost of the FLOQ server execution is approximately divided between two major components, which are the cost of splitting the binding stream B, C_s, and the cost of merging all result streams R_i, C_m. The cost of the FLOQ server execution is independent of any join strategies, i.e.:

$$C_{FLOQ} = C_s + C_m \quad (2)$$

The variables used in analysing cost models are described in Table 1.

Table 1. Variables used in the cost model

Variable	**Description**	**Variable**	**Description**
C_{Join}	Total cost of a join	C_{FLOQ}	Total execution cost in the FLOQ server
B_i	Binding stream to site *i*	C_S	Cost of splitting the binding stream *B* in the FLOQ server
R_i	Result stream from site *i*	β	A single binding from the binding stream B_i
C_i	Total cost at site *i*	C_m	Cost of merging result streams R_i in the FLOQ server
C_{σ_i}	Cost of executing σ_i in RDB_i	C_{JDBC}	Cost of JDBC call for a single binding β
$C_{\bowtie_i}$	Cost of local join at site *i*	σ_i	The query to RDB_i.
$C_{Bulkload_i}$	Cost of bulk loading in RDB_i	C_{B_i}	Cost of transferring binding stream B_i to site *i*
C_{σ_β}	Selection Cost for a single binding β	C_{R_i}	Cost of transferring result stream R_i from site *i*
RDB_i	The relational log database at site *i*	C_{Net}	Network communication overhead cost for a single binding β

The total site cost C_i is approximately divided between four major cost components: (i) transferring the binding stream B_i from the FLOQ server to the site, C_{B_i}, (ii) executing σ_i in the log database, C_{σ_i}, (iii) local join $C_{\bowtie_i}$ either in RDB_i ($C_{\bowtie_i}^{LogDB}$ for PBLJ) or in the FLOQ wrapper ($C_{\bowtie_i}^{Wrapper}$for PBJ), and (iv) transferring the result stream R_i to the FLOQ server, C_{R_i}. Thus the total site cost C_i is defined as:

$$C_i = C_{B_i} + C_{\sigma_i} + C_{\bowtie_i} + C_{R_i} \quad (3)$$

By combining equation (1), (2), and (3), the total cost of a distributed join becomes:

$$C_{Join} = C_s + C_m + max(\{ (C_{B_i} + C_{\sigma_i} + C_{\bowtie_i} + C_{R_i}) : i = 1, \ldots, n\}) \quad (4)$$

For each site, the binding stream B_i is significantly smaller than the number of logged measurements in RDB_i:

$$|B_i| \ll |Measures(RDB_i)| \quad (5)$$

For PBJ, the bind-join is performed in each FLOQ wrapper, therefore, the cost of a local join $C_{\bowtie_i}$ can be replaced with the cost of a bind-join in the wrapper, $C_{\bowtie_i}^{Wrapper}$. Also the cost of executing the sub-query σ_i that selects data from a log database, C_{σ_i}, is replaced with the *BJ selection cost*, $C_{\sigma_i}^{Wrapper}$, in the site cost in (3).

$$C_i^{PBJ} = C_{B_i}^{PBJ} + C_{\sigma_i}^{Wrapper} + C_{\bowtie_i}^{Wrapper} + C_{R_i} \quad (6)$$

In PBLJ the joins and selections are combined into one sub-query to each RDB_i. Therefore, the cost of $C_{\bowtie_i}$ and C_{σ_i} in the site cost in equation (3) for PBLJ can be replaced with the cost of join and selection in the log database ($C_{\bowtie_i}^{LogDB}$ and $C_{\sigma_i}^{LogDB}$):

$$C_i^{PBLJ} = C_{B_i}^{PBLJ} + C_{\sigma_i}^{LogDB} + C_{\bowtie_i}^{LogDB} + C_{R_i} \qquad (7)$$

In PBJ, the FLOQ server transfers the binding stream B_i to a FLOQ wrapper through the standard network protocol. Therefore, the cost of transferring bindings to each site, $C_{B_i}^{PBJ}$, is the aggregated network communication overhead for each binding, C_{Net}.

$$C_{B_i}^{PBJ} = |B_i| \times C_{Net}, \text{ where} |B_i| \geq 1 \qquad (8)$$

In PBLJ all the bindings B_i are bulk-loaded directly into the log database. The cost of sending all bindings to site i, $C_{B_i}^{PBLJ}$, is the cost of bulk loading the bindings, $C_{Bulkload_i}$.

$$C_{B_i}^{PBLJ} = C_{Bulkload_i} \qquad (9)$$

Obviously, the cost of bulk-loading in PBLJ $C_{Bulkload_i}$ is insignificant compared to sending large numbers of bindings to prepared SQL statements in PBJ:

$$C_{Bulkload_i} << |B_i| \times C_{Net}, \text{ where } |B_i| \geq 1\text{; therefore,}$$

$$C_{B_i}^{PBLJ} \leq C_{B_i}^{PBJ} \qquad (10)$$

On the other hand, the selection cost of PBLJ is also low compared to PBJ since the cost of selection performed by *RDB*$_i$ is lower than the combined cost of selection and JDBC overhead for each binding β of a binding stream B_i:

$$C_{\sigma_i}^{LogDB} \leq |B_i| \times (C_{\sigma_\beta} + C_{JDBC}), \text{ where } \beta \in B_i \text{ and } |B_i| \geq 1\text{; therefore:} \qquad (11)$$

$$C_{\sigma_i}^{LogDB} \leq C_{\sigma_i}^{Wrapper} \qquad (12)$$

Similarly, a local join in the relational DBMS is efficient compared to the join performed in a FLOQ wrapper since query optimization techniques can be applied inside a relational DBMS where the overhead JDBC calls are eliminated. Thus,

$$C_{\bowtie_i}^{LogDB} \leq C_{\bowtie_i}^{Wrapper} \qquad (13)$$

From equation (10), (12), and (13), the total cost at site i for the three components, transferring bindings (C_{B_i}), selection (C_{σ_i}), and join ($C_{\bowtie_i}$) are lower for PBLJ than for PBJ. The cost C_{R_i} of transferring the result streams R_i to the FLOQ server is equal for both PBLJ and PBJ, therefore, comparing (6) and (7):

$$C_i^{PBLJ} \leq C_i^{PBJ} \qquad (14)$$

From equation (1), as the cost of the execution at the FLOQ server C_{FLOQ} is equal for both PBJ and PBLJ, by combing equation (1) and (14) it can be stated that the overall cost of join in PBLJ is lower than PBJ:

$$C_{PBLJ} \leq C_{PBJ} \qquad (15)$$

3.2 Discussion

According to equation (15), PBLJ should always outperform PBJ in every experiment when $|B_i| \geq 1$. Equation (8) and (11) suggest that PBLJ will perform increasingly better than PBJ when scaling the number of bindings $|B_i|$. It is evident from equation (4) that, independent the chosen join strategy, when the size of the result stream $|R_i|$ is large, the tuple transfer cost (C_{R_i}) will be a major dominating component in the cost model. Therefore, the performance trade-offs between respective join strategies, are more significant when the number of tuples returned from the log database is small.

To conclude, according to the cost model, the performance evaluation should be investigated by (i) varying the number of tuples returned from the sites, (ii) scaling the number of sites, and (iii) scaling the number of bindings from the meta-database.

4 Performance Evaluation

We compared the performance of the join strategies PBJ and PBLJ based on the queries Q1, Q2, Q3, and Q4. In our real-world application each log database had more than 250 million measurements from sensor readings, occupying 10GB of raw data. The following scalability experiments were performed on six PCs (with 4 processors and 8GB main memory) running Windows 7 while: (i) scaling the number of result tuples $|R_i|$; (ii) scaling the number of sites, n; and (iii) scaling the number of bindings $|B_i|$.

Scaling the number of result tuples

Fig. 9(a) shows the execution times of Q1 for the two join strategies over a single log database, while scaling the number of result tuples $|R|$ by adjusting th. As expected from equation (12), PBLJ performs better than PBJ. Since there is only one site, PBJ is equivalent to BJ.

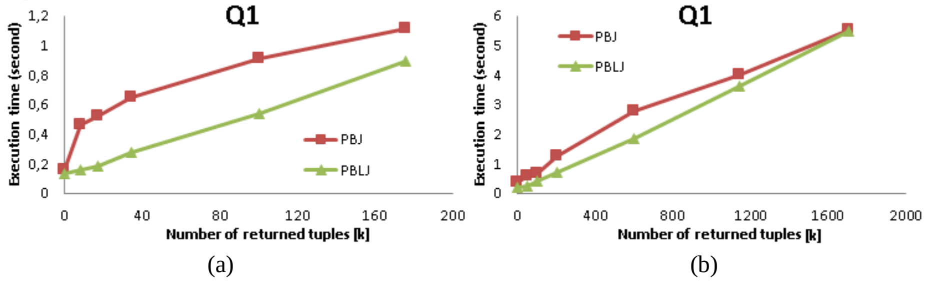


Fig. 9. Q1 (a) with one log database and (b) with six log databases

Fig. 9(b) compares the performance of Q1 for six log databases while scaling $|R|$. As expected PBLJ scales better than PBJ. However, as more tuples are returned from the log databases the network overhead is becoming a major dominating factor, making the performance difference of the join strategies insignificant. Notice that the number of returned tuples remains the same for both strategies; thus the network overhead is equal. However, PBLJ will always perform better (even with a small fraction) than PBJ since other overhead is larger for PBJ.

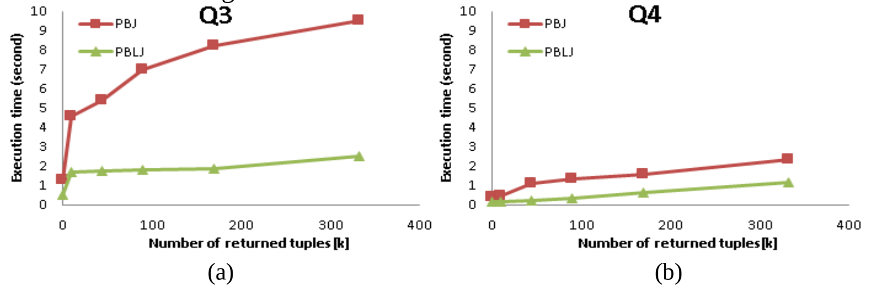


Fig. 10. Execution time for Q3 and Q4 with six log databases

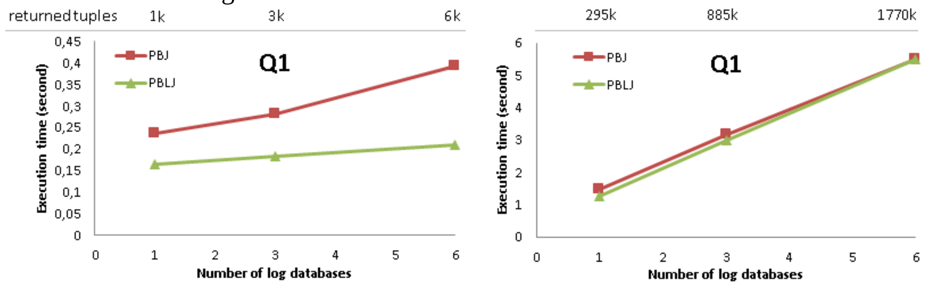
Fig. 10 compares PBJ and PBLJ for Q3 and Q4 for six log databases. Q3 is an example of a slow numerical query requiring a full scan of *Measures*, whereas Q4 is faster since it exposes the index on *Measures.mv* for query Q3. It is evident from Fig. 10 that PBLJ performs better than PBJ for both query Q3 and Q4. Fig. 10(b) shows the performance improvement due to index utilization compared to sequential scan in Q3.

To conclude, PBLJ performs better than PBJ when the number of returned tuples is increased, as also indicated by equation (15) of the cost model.

Scaling the number of log databases

Fig. 11 compares PBJ and PBLJ for Q1 when scaling the number of log databases. In Fig. 11(a) and Fig. 11(b) the total number of tuples returned from a single log database $|R_i|$ is 1K and 295K, respectively. Notice that the total number of tuples returned $|R|$ in each figure is multiplied with the fixed $|R_i|$ from each log database.

In Fig. 11(a) $|R|$ is small, so the performance difference between PBJ and PBLJ is dominating over the network cost, while in Fig. 11(b) the higher network cost makes the difference less significant.



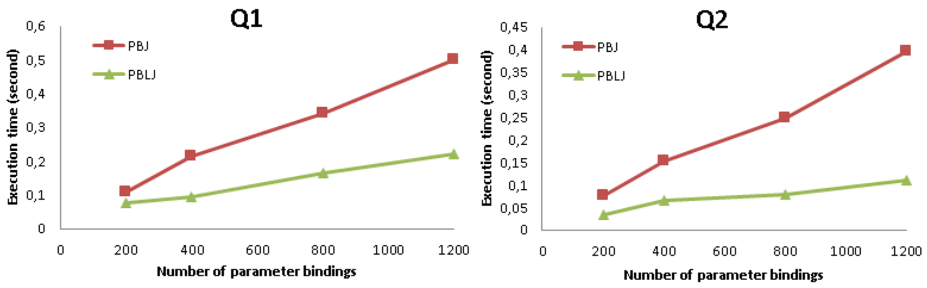
(a) 1k tuples from each database (b) 295k tuples from each database

Fig. 11. Execution time for Q1 varying number of log databases and selectivity

In summary, the overall performance of PBLJ is always better while scaling number of log databases compared to PBJ.

Scaling the number of bindings

This experiment investigates the performance of PBJ and PBLJ while varying the number of bindings $|B_i|$ from the meta-database. Fig. 12 shows the execution times for Q1 and Q2 for PBJ and PBLJ for a single log database.



(a) (b)

Fig. 12. Execution time for Q1 and Q2

From Fig. 12(a) it is evident that PBLJ performs significantly better while scaling $|B_i|$. The reason is that in PBJ, the FLOQ wrapper is performing $|B_i|$ bind-joins, so the overhead of the JDBC calls is multiplied with $|B_i|$. In all experiments the extra time for the bulk loading was less than 50ms irrespective of number of bindings $|B_i|$. This makes it insignificant for this small number of bindings relative to the size of the log databases. This confirms equation (8) and (11) of the cost model that PBJ will not scale compared to PBLJ when increasing the number of bindings. The experimental results of query Q2 that returns a single tuple per site are shown in Fig. 12(b). The reason of the better scalability of PBLJ than for Q1 is because the network communication overhead C_{R_i} in equation (4) is negligible since only one tuple is returned from each site.

In all experiments, the PBLJ join strategy performs better than PBJ, in particular while scaling the number of bindings $|B_i|$. This confirms equation (15) in the cost model. The performance improvement is more significant when the number of tuples returned from each log database is low.

5 Related work

Bind-join was presented in [4] as a method to join data from external databases [7]. We generalized bind-join to process in parallel parameterized queries to dynamic collections of autonomous log databases. Furthermore we showed that our bulk-load join method scales better in our setting.

In Google Fusion Tables [3] left outer joins are used to combine relational views of web pages, while [6] uses adaptive methods to join data from external data sources. In [9] the selection of autonomous data sources to join is based on market mechanisms. Our case is different because we investigate strategies to join meta-data with data from dynamic collections of log databases without joining the data sources themselves.

Vertical partitioning and indexing of fact tables in monolithic data warehouses is investigated in [1]. One can regard our *VMeasures* view as a horizontally partitioned fact table. A major difference to data warehouse techniques is that we are integrating data from dynamic collections of autonomous log databases, rather than scalable processing of queries to data uploaded to a central data warehouse.

In [2] the problem of making views of many autonomous data warehouses is investigated. The databases are joined using very large SQL queries joining many external databases. Rather than integrating external databases by huge SQL queries, our strategies are based on simple queries over a view (*VMeasures*) of dynamic collections of external databases, i.e. the local-as-view approach [5].

A classical optimization strategy used in distributed databases [8] is to cost different shipping alternatives of data between non-autonomous data servers before joining them. By contrast, we investigate using standard DBMS APIs (JDBC and bulk load) to make multi-database joins of meta-data with dynamic sets of autonomous log databases using local-as-view.

6 Conclusions

Two join strategies were proposed for parallel execution of queries joining meta-data with data from autonomous log databases using standard DBMS APIs: parallel bind-join (PBJ) and parallel bulk-load join (PBLJ). For the performance evaluation we defined typical fundamental queries and investigated the impact of our join strategies. A cost model was used to guide and evaluate the efficiency of the strategies. The experimental results validated the cost model. In general, PBLJ performs better than PBJ when the number of bindings from the meta-database is increased.

In the experiments a rather small set of autonomous log databases were used. Further investigations should evaluate the impact of having very large number of log databases and different strategies to improve communication overheads, e.g. by compression.

Acknowledgments. This work is supported by EU FP7 project Smart Vortex and the Swedish Foundation for Strategic Research under contract RIT08-0041.

References

1. Datta, A., VanderMeer, D.E., Ramamritham, K.: Parallel Star Join + DataIndexes: Efficient Query Processing in Data Warehouses and OLAP. J. IEEE TKDE. 14(6), 1299-1316 (2002)
2. Dieu, N., Dragusanu, A., Fabret, F., Llirbat, F., Simon, E: 1,000 Tables Inside the From. J. ACM VLDB. 2(2), 1450-1461 (2009)
3. Garcia-Molina, H., Halevy, A.Y., Jensen, C.S., Langen, A., Madhavan, J., Shapley, R.,Shen, W.: Google fusion tables: data management, integration and collaboration in the cloud. In: SoCC, pp. 175-180 (2010)
4. Haas, L., Kossmann, D., Wimmers, E., Yang, J: Optimizing queries across diverse data source. In: VLDB, pp. 276-285 (1997)
5. Halevy, A., Rajaraman, A., Ordille, J.: Data Integration: The Teenage Years. In: VLDB, pp. 9-16 (2006)
6. Ives, G., Halevy, A., Weld, D.: Adapting to Source Properties in Processing Data Integration Queries. In: SIGMOD, pp. 395-406 (2004)
7. Josifovski, V., Schwarz, P., Haas, L., Lin, E.: Garlic: A new flavor of federated query processing for DB2. In: SIGMOD, pp. 524-532 (2002)
8. Kossmann, D.: The State of the Art in Distributed Query Processing. J. ACM Computing Surveys. 32(4), 422-469 (2000)
9. Pentaris, F., Ioannidis, Y.: Query Optimization in Distributed Networks of Autonomous Database Systems. J. ACM Transactions on Database Systems. 31(2), 537-583 (2006)
10. Risch, T., Josifovski, V.: Distributed Data Integration by Object-Oriented Mediator Servers. J. Concurrency and Computation: Practice and Experience. 13(11), 933-953 (2001)
11. Smart Vortex Project, http://www.smartvortex.eu/
12. Truong, T., Risch, T.: Scalable Numerical Queries by Algebraic Inequality Transformations. In: DASFAA, pp. 95-109 (2014)
13. Zhu, M., Risch, T.: Querying combined cloud-based and relational databases. In: CSC, pp. 330-335 (2011)
14. Zhu, M., Stefanova, S., Truong, T., Risch, T.: Scalable Numerical SPARQL Queries over Relational Databases. In: LWDM workshop, pp. 257-262 (2014)