A query optimizer based on a full reducer algorithm for SuperSQL

Arnaud WOLF†, Kento GOTO†, and Motomichi TOYAMA††

† ††Department of Information and Computer Science,
Keio University
Hiyoshi 3–14–1, Kouhoku-ku, Yokohama-shi, Kanagawa, 223–8522 Japan
E-mail: †{arnaud,goto}@db.ics.keio.ac.jp, ††toyama@ics.keio.ac.jp

Abstract SuperSQL is an extension of SQL that automatically formats data retrieved from the database into various kinds of application data as an output of a query. Output formats include, but are not limited to, HTML, HTML5, XML and PDF. SuperSQL continuously evolves to support current emerging technologies particularly related to web development. Current developments lead us to identify improvement points and remodel the design of the SuperSQL architecture. Specifically, the current implementation includes useless cartesian product that significantly slows down the execution of the queries, because all the tables are retrieved together, while we could actually retrieve them independently from each other. This paper proposes an optimizer by query decomposition based on a full reducer algorithm.

Key words Data retrieval, query optimization, SuperSQL, query language

1. Introduction

SuperSQL is an extension of SQL that enables to generate various kinds of application data directly as a result of a query. Its syntax is similar to SQL with additional formatting capabilities. Possible application data output formats include HTML, PDF, XML, XLS, Ajax, etc. The current main usage of this language is the generation of websites or web applications, with the advantage not having to use any other language that would require more programming skills. A sample of SuperSQL query with its resulting webpage is shown in Figure 1.

```
GENERATE HTML [
      {"Books"![b.title]!
      {"Authors"![a.name]!},
      {"Publishers"![p.publisher]!}
  ]!
FROM books b, publishers p, authors a
```

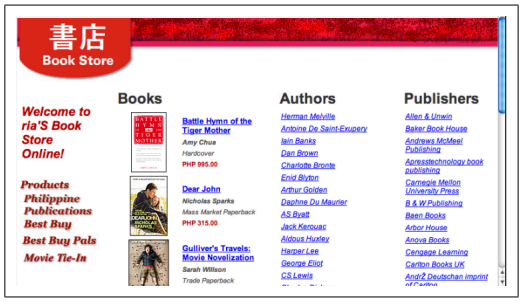

Figure 1 Sample SuperSQL query and its result

If many features enabling a richer and more flexible utilisation of SuperSQL have been developed recently, SuperSQL also faces important performance issues. More specifically, the structure in the compiler that is responsible of data retrieval roughly generate the SQL query by gathering all the involved tables within it. This causes the generation of very big intermediate tables, slowing down the execution time.
Our current work consists in developing an optimizer that will split the original SQL query into several SQL queries by using query decomposition, in order to reduce the size of the intermediate tables. Our query decomposition model itself is based on the concept of full reducers, an old optimization technique often used in distributed databases.
The outline of this paper is as follow. In part 2, we introduce an simple experience highlighting the performance issues of SuperSQL. In part 3, we introduce more in details SuperSQL, in term of architecture and data representation. In part 4, we introduce the current optimizer implementation and part 5 is the conclusion.

2. Performance issues in SuperSQL

A very simple experiment shows the performance issues of the current implementation. The experience is based on the following query:

GENERATE HTML
[p.name]!, [s.name]!
FROM professor p, student s
WHERE p.dept = 'ICS' AND s.dept = 'ICS'

This query writes in an HTML file both lists of professors and students from the department ICS. While varying the

size of both tables professor and student, we executed the above query and collected the execution time. For a given size, we also executed the query that writes only the list of professors, and the one that writes only the list of students, and compare the sum of execution time of those queries to the one of the original query. The results are shown in Figure 2.

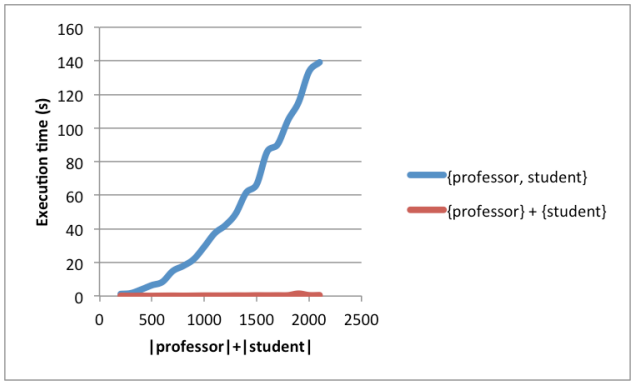

Figure 2 Result of the experiment

The results clearly shows that the original query induces a behaviour that is far from the expected trend, namely, the sum of the trends of the individual queries. This lead us to understand that we should manage to change the behaviour of the SuperSQL compiler, in order it to split the retrieval of the two tables.

3. SuperSQL inner architecture and process

As specified in the introduction, the syntax of SuperSQL is very similar to SQL. However, in addition, the user also has to specify how he wants the data to be organised in the generated output file. The structure of a SuperSQL query is the same as the one of a SQL query, except that the SELECT clause is replaced by another structure. A SuperSQL query always starts by GENERATE "format", where "format" is the desired format of the output file, and is followed by the **Target Form Expression (TFE) clause**, whose role is to organised the future data into a tree structure. We call the remaining part of the query the **SQL clause**. The structure derived from the TFE clause is called the **schema**.

Figure 3 shows the architecture of the SuperSQL compiler. The SuperSQL is first parsed, and two parts are extracted, the schema (from the TFE clause), and components, namely the table names, predicates and so on. The components are sent to the **SQL query maker** that generates the SQL query aimed at retrieving the data. The extracted data are then sent to the **data constructor**, that organises the data accordingly to the schema. Then the **code generator** generates the output files.

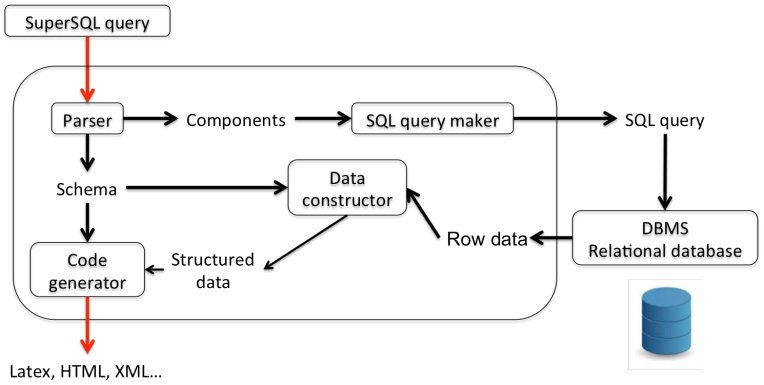

Figure 3 Architecture of the SuperSQL compiler

TFE's semantic is based on operators. The two main operators are the connectors and the repeaters. Connectors are used to connect the data within a specific dimension. Especially, in case of HTML files, there are three dimensions: the horizontal dimension represented by a comma (","), the vertical dimension by an exclamation point ("!") and the link dimension by a percent symbol ("%").

The repeaters, symbolised by a pair of brackets ("[]"), connects the components written inside it in its associated dimension, and will unroll all the tuples of the corresponding relation.

From this semantic, a tree structure, called the **TFE tree**, representing the hierarchy of the future retrieved data, is derived. Figure 4 shows an example of query and its tree structure.

GENERATE HTML
[a.name, [b.name]!, [c.name]!]!
FROM A a, B b, C c

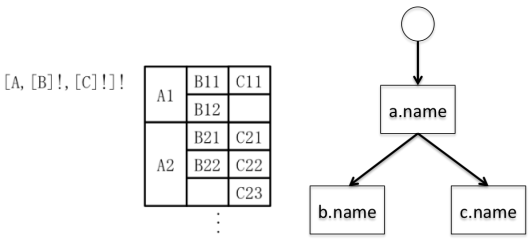

Figure 4 Sample SuperSQL query and its tree structure

If two components are not embedded within the same repeaters, they are called **independent**, that is, they don't form a tuple together. In case of our example, b and c are independent, but a and b or a and c are not independent. Therefore, a **branch** of the TFE tree is the largest type of set of components that are all dependent each other

In the current implementation, the SQL query maker derives

the SQL queries just from the components issued from the SQL clause. It doesn't use the schema. The SQL clause, containing FROM clause, WHERE clause and so on, is just copied in the final SQL query, and the SELECT clause gathers all the desired attributes. For instance, the SuperSQL query in Figure 4 leads to the following SQL query:

SELECT a.name, b.name, c.name
FROM A a, B b, C c

The result of executing this query is the cartesian product between the tables A, B and C, which can be a very huge table. However, the desired organisation specified by the schema doesn't expresses a cartesian product, because tables B and C are here independent. To be more concrete, if each tables contains 30 tuples, the size of the intermediate table can reach almost 4000 tuples, while the size of the array in the output file would not overcome 30 rows. Therefore, a post-process is performed in order to remove useless tuples and get the final structure.

If the intermediate table retrieved from the cartesian product is very huge, the retrieval and the post-process operations cause huge overhead. Thus, the purpose of the optimizer we are developing is to reduce the size of this intermediate tables by splitting the SQL query into several SQL queries involving only tables that are dependent each other considering the schema.

The tree structure used in the optimizer described in this paper is derived from the TFE tree above. For our optimizer, we just need to know the tables in themselves, and not necessarily the attributes (at least, not at this stage). Therefore, our tree structure is similar to the above one, except that we replace each component by its table's name, and remove the duplicates in each node. Moreover, in case a table appears more than one time in a branch, we only keep its uppermost occurence in the branch. Figure 5 shows the modified tree of the example in Figure 4.

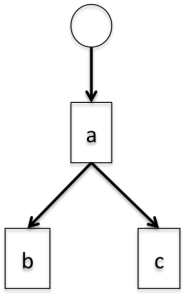

Figure 5 The modified TFE tree

4. Optimizer based on query decomposition and full reducer

4.1 Principle of query decomposition

In this section, we introduce the general principle of query decomposition we designed under the logic of SuperSQL. The inputs and outputs of the query decomposition process are specified in Figure 6. The input is a SQL query from where

Inputs	
R	Set of relations. A relation is a pair {name, alias}
F	Algebraic expression
P	Predicate
B= {B1,...,Bn}	Branches: disjoint sets of relations of **R** aimed to be displayed together

Outputs	
Pa={**B1**,...,**Bm**}	Partition of B
F = F1,...,Fm	Algebraic expression over partitions of Pa
P = P1,...,Pm	Predicates over partitions

Figure 6 Inputs and outputs of the query decomposition process

we extract a set of tables and an algebraic extraction from the FROM clause, and a predicate from the WHERE clause. The purpose of query decomposition is to find a partition of the set of branches that optimizes the data retrieval, and for this partition, to find the appropriate set of algebraic expressions **F** and the set of predicates **P** satisfying the constraint equation (1). Equation (1) expresses that the result obtained for each branch from each partition should be equivalent to the projection on this branch of the result obtained from the original query.

$$\forall i \in [1,m], \forall B \in \mathbf{B_i}, \pi_B(\sigma_{P_i}(F_i(\mathbf{B_i}))) \equiv \pi_B(\sigma_P(F(\mathbf{B})) \quad (1)$$

4.2 Principle of full reduction

Let us **Pa** be a given partition of **B**. In many cases, it is possible to retrieve all the sets from **Pa** with a unique query for each set. However, there are also many cases where it is not possible, because the expression F or the predicate P of the original query involves a dependance between some branches. As for an example, let us consider the following query :

GENERATE HTML
[a.name]!, [b.name]!, [c.name]!
FROM A a, B b, C c
WHERE a.id = b.id AND b.reg = c.reg AND a.dept = "ICS"

In case of this query, we have three branches, but the WHERE clause involves predicates between them. Therefore, if we retrieve separately each of the three branches, we still need to check, afterward, if each tuple of each relation satisfies the condition with at least one tuple in the another relation.

[2] designs an algorithm of full reduction destined to distributed database systems. In a distributed database system,

a query may involve relations in different locations, and the data retrieved from each of those locations has to be gathered in a common place before being treated. Therefore, in order to avoid overhead due to intersite communication, it can be desired to reduce the size of data from each location before joining them. In case of the above example, if A, B and C are stored in different locations, it could be useful to first select, for instance, only the tuples of A that satisfies the predicate with at least one tuple of B, before actually performing the join.

[2] introduces a technique to reduce as much as possible the size of relations by performing what they call a full-reducer sequence, applicable on simple queries (involving only inner joins and no special algebraic operators). It consists on the following steps. At first, given a sql query q, we construct a graph $\mathbf{G_q} = \{\mathbf{R}, \mathbf{E}\}$, where $\mathbf{R}$ is a set of relations, and $\mathbf{E}$ a set of pairs of relations such as, for a pair (R_1, R_2) of $\mathbf{R}$, q includes a binary predicate involving both R_1 and R_2. We call this graph the query graph of q.

The second step is to perform a recursive sequence of updates, called full reducer, that reduces the size of the desired table by keeping only tuples that satisfies all the predicates with at least one tuple from other relations it is connected with. As this sequence is described in [2], we won't describe it in details here.

An important point is that, according to [2], we can find a full reducer if and only if the query graph does not contain cycle, or namely, is a tree. [1] describes a way to reduce the size of relations even if the graph contains cycles, but it cannot be applied in the context of SuperSQL. As a matter of fact, in case of SuperSQL, the purpose is not to finally join the relations, but to retrieve the exact amount of information from each branch without joining it, and then we need a full reducer. And [2] proved that if the query graph of q contains cycle, it is impossible to find a full reducer for q. The solution of [1] enables only to partially reduce the size.

In case of SuperSQL, the vertices of the query graph are not relations, but branches, and there is an edge between two branches if there exists a binary predicate involving tables from both branches. Building such a graph requires a pretreatment on predicates that is described later.

As an example, let us consider the above example. The three branches derived from this examples are B_1 = (A, a), B_2 = (B,b) and B_3 = (C, c). Figure 7 is an example of query tree under the context of SuperSQL, and Figure 8 is the full-reducer sequence that enables to retrieve the data from each branch.

4.3 Model and limitations of the optimizer

The optimizer we are currently developing has two main limitations. The first one is that it can only be applied on

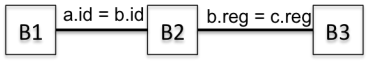

Figure 7 Example of query tree

```
1. CREATE TABLE tmp1 as SELECT * FROM A WHERE dept = "ICS"
2. CREATE TABLE tmp2 as SELECT * FROM B
3. CREATE TABLE tmp3 as SELECT * FROM C
4. REMOVE FROM tmp2 b WHERE NOT EXISTS (SELECT * FROM
     tmp3 c WHERE c.reg = b.reg)
5. REMOVE FROM tmp1 a WHERE NOT EXISTS (SELECT* FROM
     tmp2 b WHERE b.id = a.id)
6. SELECT a.name FROM tmp1 a
7. SELECT b.name FROM tmp2 b WHERE EXISTS (SELECT* FROM
     tmp1 a WHERE a.id = b.id)
8. SELECT c.name FROM tmp3 c WHERE EXISTS (SELECT* FROM
     tmp2 b WHERE b.reg = c.reg)
```

Figure 8 The full reducer of above example

queries that does not include complex operators, that is, queries that only involves inner joins in the FROM clause (e.g., no outer joins), no aggregates, and no HAVING clause in the WHERE clause. Especially, the generalisation on any FROM clause, even the one including outer joins, will be the object of near future works. The second one is about the predicate. We will see below that we need to factorize the predicate, in order to make the queries. If the factorization is not possible, then the optimization is also impossible.

Considering those limitations, under the scope of the optimizer, a SuperSQL query can be defined with the following inputs:

- a set of relations $\mathbf{R}$.
- a set of branches $\mathbf{B}$, with each element of $\mathbf{B}$ being a subset of $\mathbf{R}$.
- a predicate P.

Predicates are modeled as polynomials of elementary predicates. The + operator corresponds to OR and the × operator corresponds to AND.

$$P = \sum_{i=0}^{p} (\prod_{j=0}^{m_i} p_{ij}) \tag{2}$$

p_{ij}'s are called elementary predicates. Namely, it is the predicates that are directly found in the WHERE clause, such as "a.id = b.id". An elementary predicate can either be a unary predicate if it involves only one relation, or a binary predicate if it involves two relations.

4.4 The different steps of the optimization

In this section, we describe the different steps of the optimization performed by the optimizer we are developing.

4.4.1 Pretreatment of branches

As mentioned in Figure 6, in order to be usable as an input of the query decomposition process, the set of branches should be disjoint. However, in general, it is far from being the case. The tree represented in Figure 5 includes two

branches: $\mathbf{B_1}$ = {(A, a), (B, b)} and $\mathbf{B_2}$ = {(A,a), (C,c)}, and the relation (A, a) is a common relation of the two branches.

In order to make the branches distinct, we replace each common relation of the two branches by relations with same name, but different aliases. As for the example, the two branches become $\mathbf{B_1}$ = {(A, a1), (B, b)} and $\mathbf{B_2}$ = {(A,a2), (C,c)}.

Let us assume that the predicate P of this query is defined as being:

$$P = (a.v = b.v) \times (a.w = c.w) \tag{3}$$

Then the alias in each of the elementary predicate is replaced by the alias of the specific branch, and a binary predicate specifying that the primary key of a1 and a2 should be equal is added.

$$P' = (\prod_{pkey \in \mathbf{K_A}} (a1.pkey = a2.pkey)) \times (a1.v = b.v) \times (a2.w = c.w) \tag{4}$$

where $\mathbf{K_A}$ is the set of primary keys of A. The set of relations having been duplicated like that is stored in prevision of data construction.

4.4.2 Pretreatment of predicates

As the vertices of the query graph are branches and not relations, we need to build it with predicates defined per branches, and not per relation. Thus, we need to extract those branches predicate from the predicate P'. However, in order to do that, we need to transform P' into a specific shape.

$$P' = \prod_{B \in \mathbf{B}} P_B \prod_{(B,B') \in \mathbf{B}^2} P_{B,B'} \tag{5}$$

So the second step of the optimizer is to factorise the predicate P' in order it to fit with equation (5). The P_B's are used in queries materializing each branches, and the $P_{B,B'}$ are used to build the query tree and perform the full reduction.

However, there may be some predicates that cannot be factorised as above. For instance, still considering the example of Figure 5, the predicate of the equation (6) cannot be factorised as above. In such a case, the optimization is stopped, and the original retrieval within a single query is performed.

$$P' = (b.v = c.v) + (b.w = c.w) \tag{6}$$

In order to check if a predicate is factorisable, we extract all the elementary predicates of P and put it in the appropriate monomial (the one involving its branch(es)), create a polynomial that is the product of all those monomial, and check the equality by identification. In case of the predicate of equation (6), for instance, the inequality (7) shows that P' is not factorisable.

$$(b.v = c.v) \times (b.w = c.w) \neq (b.v = c.v) + (b.w = c.w) \tag{7}$$

4.4.3 The query graph and the management of cycles

If the factorisation succeeded, the $P_{B,B'}$'s are used in order to build the query graph, that is, if for a pair of branch (B, B'), there is a predicate $P_{B,B'}$, the branches B and B' are connected together in the graph.

Then the optimizer checks if the graph contains cycles or not. If the graph does contain cycles, the partition is rejected. However, unlike the case of factorisation failure, the opimization does not stop completely. Rather, we extract from the graph a cycle base (a set of set of vertices of the graphs, each of them representing a cycle, and such as every cycles of the graph are combination of those cycles). Then we change the partition **Pa** in order to eliminate all the cycles of the base, that is, by joining all the branches in a cycle together.

Figure 9 shows an example of case including cycles.

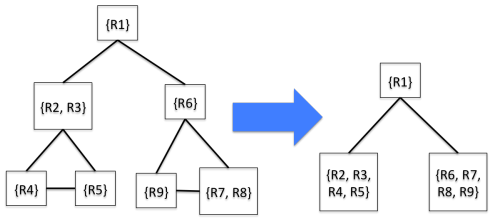

Figure 9 Example of processing in case of tree including cycles

4.5 Data retrieval

When a query graph without cycles is obtained, the full reduction sequence can be performed, as it is described in above section. Then the queries are generated and executed in the appropriate order, and the result sets are extracted.

At this stage, an simple but fundamental step has to be performed. The query graph is built on either a binary predicate connects two branches or not. However, even if there is no binary predicates connecting two branches, there is still a dependence between them, because under the limitation of this optimizer, we assume that only inner joins connects the relations.

Figure 10 illustrates a case where we need to perform an additional treatment. As a matter of fact, in this case, the branch B4 is not connected to any other branch. Therefore, all the data of B4 is retrieved without going through any full-reducer sequence. However, if any branch among B1, B2 or B3 issues an empty set, in order the constraint equation 1 to be satisfied, the final result set of B4 should also be empty. Thus, the check of emptiness is an compulsory operation to perform after the retrieval. In a general case, the checking

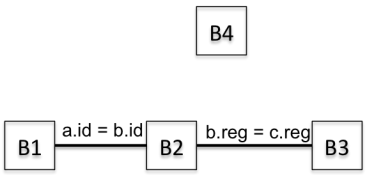

Figure 10 Example of case where emptiness check is necessary

of emptiness has to be perform in order to ensure that the emptiness of the result set of one connected component of the query graph causes the emptiness of all the other connected components of the graph.

4.6 Data construction

The last step of the process is data construction. In this stage, the data is finally organised in a tree structure corresponding to the schema. Figure 11 shows an example of output structure of the data constructor.

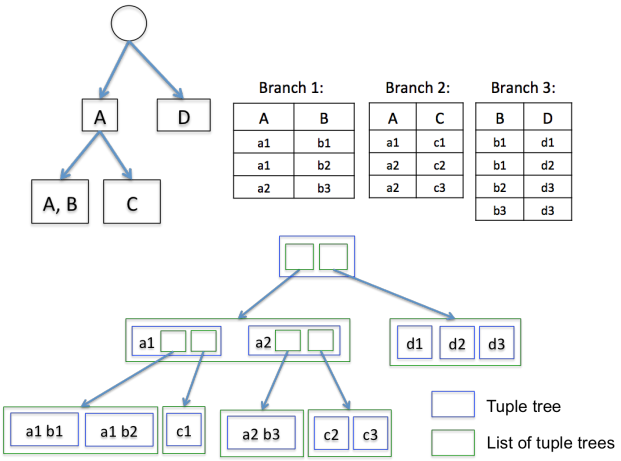

Figure 11 Example of constructed data structure

The output structure includes a sub-structure called tuple tree. A tuple tree is a list of elements that are either data or another list of tuple tree. The root of the final structure is a single tuple tree, containing only the list of tuple trees for each root of the schema.

The data are input in the structure branch after branch. For a given branch, data are input tuple after tuple. For the insertion of a given tuple, the structure is traversed recursively considering the schema. When considering a given node of the tree, the existence of a tuple tree containing the values of the given tuple is checked. Then if necessary, a new tuple tree containing those values is created. The same process is applied recursively to the subtrees of this tuple tree.

As this step requires to manipulate the entire set of data retrieved from the database, the smaller the intermediate tables are, the faster the execution is.

The optimizer stands on a heuristic: we assume that the more the original query is divided, the faster the execution is. As a matter of fact, the decomposition necessarily leads to smaller intermediate tables. The weak point of this heuristic is that the data retrieval, with materialization and multiple update during full reduction, may cause some important overhead. However, our assumption stands on the hypothesis that users of SuperSQL don't often write queries with a great number of branches, or with complex predicates. Therefore, we assume that the query graph won't cause long full-reduction chains, and that the overhead due to the data retrieval will be compensated by the gain on performance on data-construction. All those assertions will be verified on near future works.

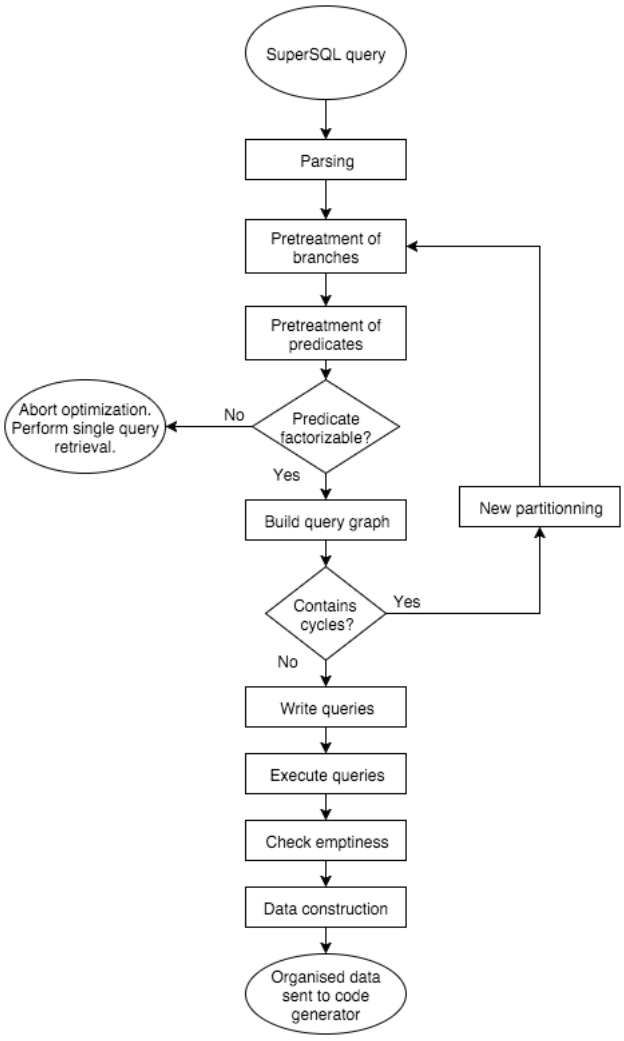

Figure 12 Flowgraph of the optimizer

5. Conclusion and future works

In order to face the issues of performance due to undesirable cartesian product during the data retrieval, we designed an optimizer based on query decomposition and full-reduction. Figure 12 shows the flowgraph of all the steps of

optimization.

We adapted the full-reduction principle widely used in distributed database for the purpose of SuperSQL. The developed optimizer has limitations concerning the complexity of queries, and currently accepts only queries excluding outer joins, having clause and aggregates. The optimizer has not been properly evaluated yet, but the evaluation of this limited version is the next stage of our work. The evaluation will include, the evaluation of gain of performances provided by the optimizer, the measurement of the overhead brought by the optimization ourselves, and the validity of our heuristic, that is, considering that users mostly writes queries that tends to be executed more efficiently with our optimizer.

Near future works will also involves the generalisation of the query decomposition model to algebraic expressions including outer joins, that will strongly rely on the notion of independence. This step requires deeper thinkings related to relational algebra.

References

[1] Bernstein, Philip A., and Nathan Goodman. "Power of natural semijoins." SIAM Journal on Computing 10.4 (1981): 751-771.

[2] Bernstein, Philip A., and Dah-Ming W. Chiu. "Using semijoins to solve relational queries." Journal of the ACM (JACM) 28.1 (1981): 25-40.

[3] Toyama, Motomichi. "SuperSQL: an extended SQL for database publishing and presentation." ACM SIGMOD Record. Vol. 27. No. 2. ACM, 1998.

[4] Borromeo, Ria Mae, and Motomichi Toyama. "Optimization of SuperSQL Execution by Query Decomposition."

[5] Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. Database system concepts. Vol. 4. Singapore: McGraw-Hill, 1997.