# Parallelizing Computations of Full Disjunctions

Matteo Paganelli[a], Domenico Beneventano[a], Francesco Guerra[a], Paolo Sottovia[a,b]

[a] *DIEF-UNIMORE, via Vivarelli 10, 41125 Modena, Italy*
[b] *DISI-UNITN, via Sommarive 9, 38123 Povo (TN) Italy*

**Abstract**

In relational databases, the full disjunction operator is an associative extension of the full outerjoin to an arbitrary number of relations. Its goal is to maximize the information we can extract from a database by connecting all tables through all join paths. The use of full disjunctions has been envisaged in several scenarios, such as data integration, and knowledge extraction. One of the main limitations in its adoption in real business scenarios is the large time its computation requires. This paper overcomes this limitation by introducing a novel approach PARAFD, based on parallel computing techniques, for implementing the full disjunction operator in an exact and approximate version. Our proposal has been compared with state of the art algorithms, which have also been re-implemented for performing in parallel. The experiments show that the time performance outperforms existing approaches. Finally, we have experimented the full disjunction as a collection of documents indexed by a textual search engine. In this way, we provide a simple technique for performing keyword search over relational databases. The results obtained against a benchmark show high precision and recall levels even compared with the existing proposals.

*Keywords:* Full Disjunction, Parallel Computing, MapReduce

## 1. Introduction

Due to their capability of managing and storing data in an effective and efficient way, relational databases have been largely adopted in business applications. The relational database design methodology, based on normal forms, assures data integrity and eliminates redundancy by coding the information into a number of tables connected with each other via foreign key relationships. Nevertheless, in several scenarios, the fragmentation induced by the model may represent a big obstacle for a user to understand and work with the entire database content. The *universal relation assumption* [1] allows users to address this kind of problems, by treating data if it were all in a single relation over all the attributes. The universal relation computation requires data in different relations to be in some way integrated, and, to do it, users have to know how the tables are connected

*Email addresses:* matteo.paganelli@unimore.it (Matteo Paganelli), domenico.beneventano@unimore.it (Domenico Beneventano), francesco.guerra@unimore.it (Francesco Guerra), paolo.sottovia@unitn.it (Paolo Sottovia)

in the database. Let us suppose that we want to apply the universal relation assumption to a database composed of two relations. The universal relation can be obtained through a simple straightforward integration process that generates a relation schema composed of the union of the attributes of the input relations. The universal relation is populated by means of the application of the outerjoin operator (usually on the attributes either sharing the same names – via natural outerjoins, or specified in the foreign key relations – via equijoins), which avoids loss of data from the source tables.

The population of the universal relation of a database composed of several tables requires particular attention. The ourterjoin operator is not associative: its application may generate different results if we consider a different order in the tables involved in the join paths. Moreover, there is not a unique order: the tables can be linked through different paths and cycles may arise. Each path conveys different semantics and a cycle can be transformed in a number of paths, one for each involved table.

The *full disjunction* [2] has been proposed to cope with these issues. It consists of an associative extension of the full outerjoin to an arbitrary number of tables completely preserving the entire information content of the data source. This operator is implemented by joining tuples over all possible paths connecting the database tables, thus making its computation a critical task. As described in Section 4, a number of algorithms implementing the operator have been proposed in the literature. Nevertheless, they have proven to be inadequate in real scenarios for dealing with large data sources, due to the execution time required.

The full disjunction is of paramount importance in all scenarios where a de-normalization of relational databases completely preserving the information is needed. The existing tools able to compute the full disjunction are not easily usable in real environments, due to the computational complexity of the algorithms implemented and the long execution times they require. If we were able to provide an efficient computation, this would have a big impact in a large number of scenarios. A typical scenario is data integration. Different databases can model the same real-world domain in different ways. De-normalizing the data before the integration eases the process. Another interesting scenario is provided by Data Mining and Machine Learning, where the input is typically constituted by a single table. Data in relational databases has to be de-normalized to be used with these approaches. The recent research focused on Big Data made available efficient data abstractions/structures (e.g., RDD[3]) and MapReduce based frameworks for supporting parallel computation on massive datasets (e.g., Apache Hadoop[1], Apache Spark[2]).

In this paper, we leverage such technical advances by introducing PARAFD (PARAllel Full Disjunction): an approach providing an efficient implementation of the full disjunction. Our proposal divides the computation into different phases: a) creation of a database graph representing the database schema; b) computation of all spanning trees over the database graph; c) computation of a full disjunction for each spanning tree; and d) merging the full disjunctions by removing duplicated and subsumed items. The advantages of this proposal mainly lie in the availability of optimized and low complexity algorithms for spanning trees computation and in a novel parallel implementation of a multi-relation hash star join algorithm able to reduce the overhead resulting from the

[1] http://hadoop.apache.org/
[2] http://spark.apache.org/

distribution of data on the network. Moreover, there are scenarios where we do not need a "complete" full disjunction including all combinations of tuples from the database tables, but only the most significant ones, according to some quality metrics. This would reduce the computation time. PARAFD can be adapted for creating an *approximate full disjunction*, thanks to the definition of a measure (based on the Pointwise Mutual Information in our implementation) for identifying the "most significant" spanning trees, and computing the full disjunctions associated to them only.

We performed a deep experimentation of PARAFD, by also comparing it against two existing algorithms: $IncrementalFD$ [4] and $BiComNLOJ$ [5]. Since it could be unfair to compare parallel and sequential algorithms, we extended and reimplemented both the approaches so that they can perform in parallel. Four variants of $IncrementalFD$, with different levels of parallelization, are presented in Section 4. The experiments highlight the efficiency of our proposal, reducing the time required for generating all full disjunctions up to 4 magnitude orders. The effectiveness of PARAFD has been evaluated in the challenging scenario of keyword search over relational databases. We considered the full disjunction as a collection of documents (one for each tuple composing it) to be indexed by a text retrieval engine. We experimented this search system with a well known benchmark [6] obtaining results with high precision levels.

Summarizing, the main contributions of this paper are:

- the development and implementation of an algorithm based on spanning trees and a parallel implementation of a multi-relation hash join algorithm for computing the full disjunctions of a database;
- the re-design of the $IncrementalFD$ and $BiComNLOJ$ algorithms to be able to perform in parallel. Four implementations of $IncrementalFD$, with different levels of parallelism, are described and evaluated in the paper;
- a technique for computing an approximate full disjunction, i.e., a full disjunction with only the most significant tuples, according to a quality measure;
- a deep experimentation of the approaches with real and large datasets showing that PARAFD outperforms the state of the art.

The rest of this paper is organized as follows: Sections 2 and 3 formally define the full disjunction operator and introduce PARAFD for its computation. In Section 4, we describe two main existing techniques for computing the full disjunction and introduce four parallel implementations. Related work is discussed in Section 5. The experimental evaluation is presented in Section 6 and finally in Section 7 we sketch out some conclusion and future work.

## 2. Preliminaries

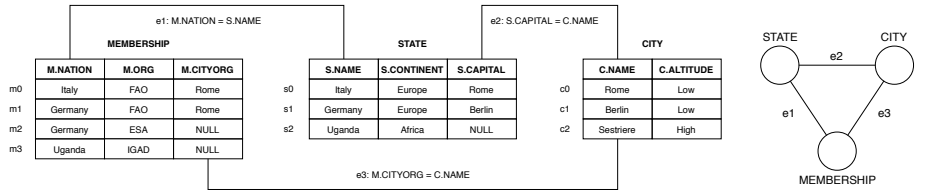
The full disjunction is an associative extension of the outerjoin [2]. Approaches aiming to maximize the capability of joining pieces of data from different relations built full disjunctions upon natural outerjoins [7, 4, 5] (i.e., equijoin on common attributes). In this paper, we extend those papers by introducing a definition of full disjunction based on equijoins between foreign and primary keys. In this way, we take only into account

the connections between the tables introduced by the database designer, thus preserving the original semantics of the data[3].

Let us consider a relational database with $n$ relations $\mathcal{R} = \{R_1, ..., R_n\}$, where each relation $R_i$ has a schema $sc(R_i)$ composed of $p_i$ attributes $R_i.A_1, ..., R_i.A_{p_i}$, a primary key $PK \subseteq sc(R_i)$ and possibly multiple foreign keys $FK \subseteq sc(R_i)$ referring to other relations.

The schema of $\mathcal{R}$, denoted $sc(\mathcal{R})$, is the union of the schemas $sc(R_i)$ of relations in $\mathcal{R}$. The *schema graph* of $\mathcal{R}$, denoted $\mathcal{G}_{sc(\mathcal{R})} = (V, E)$, is an undirected graph showing connections between relations generated by foreign key relationships, where $V$ and $E$ are the set of nodes and edges, respectively. There is a node for each relation $R_i$.

There is an edge $e{=}(R_i, R_j) \in E$ between the nodes $R_i$ and $R_j$, if the primary key $R_i.PK$ defined on $R_i$ is referenced by the foreign key $R_j.FK$ defined on $R_j$. Note that, in general, there may be multiple edges between the same pair of nodes, generated by different foreign keys on the same relations. For sake of simplicity, in the following, we assume that only an edge is possible. We say that $\mathcal{R}$ is connected if $\mathcal{G}_{sc(\mathcal{R})}$ is connected.




MEMBERSHIP

| | M.NATION | M.ORG | M.CITYORG |
|---|---|---|---|
| m0 | Italy | FAO | Rome |
| m1 | Germany | FAO | Rome |
| m2 | Germany | ESA | NULL |
| m3 | Uganda | IGAD | NULL |

STATE

| | S.NAME | S.CONTINENT | S.CAPITAL |
|---|---|---|---|
| s0 | Italy | Europe | Rome |
| s1 | Germany | Europe | Berlin |
| s2 | Uganda | Africa | NULL |

CITY

| | C.NAME | C.ALTITUDE |
|---|---|---|
| c0 | Rome | Low |
| c1 | Berlin | Low |
| c2 | Sestriere | High |

(a) Content of the tables




(b) Schema graph

| | FD({C,S,M}) |
|---|---|
| fd0 | {c0, s0, m0} |
| fd1 | {c0, s0, m1} |
| fd2 | {c0, s1, m1} |
| fd3 | {c1, s1, m1} |
| fd4 | {c1, s1, m2} |
| fd5 | {s2, m3} |
| fd6 | {c2} |

(c) Full disjunction

| M.NATION | M.ORG | M.CITYORG | S.NAME | S.CONTINENT | S.CAPITAL | C.NAME | C.ALTITUDE |
|---|---|---|---|---|---|---|---|
| Italy | FAO | Rome | Italy | Europe | Rome | Rome | Low |
| Germany | FAO | Rome | Italy | Europe | Rome | Rome | Low |
| Germany | FAO | Rome | Germany | Europe | Berlin | Rome | Low |
| Germany | FAO | Rome | Germany | Europe | Berlin | Berlin | Low |
| Germany | ESA | NULL | Germany | Europe | Berlin | Berlin | Low |
| Uganda | IGAD | NULL | Uganda | Africa | NULL | NULL | NULL |
| NULL | NULL | NULL | NULL | NULL | NULL | Sestriere | High |

(d) Full disjunction relation

Figure 1: The running example

**Example 1:** Figure 1(b) is the schema graph of the database in Figure 1(a) adopted as a running example. It provides details about countries which are members of organizations located in cities. Its structure is composed of three tables: City (C), State (S) and Membership (M). The connections between the tables are coded in the database through primary / foreign key relationships.

The relational model allows users to merge data from different relations through the join operator. The *j*oining tree of tuples is the data structure that has been introduced in the literature [8] to represent tuples connected by a join operation.

[3]Note that, by introducing proper join conditions, our formalization can be used to represent full disjunctions based on natural outerjoins.

**Definition 1** (**Joining trees of tuples**). *Given a database* $\mathcal{R} = \{R_1, ..., R_n\}$ *with schema graph* $\mathcal{G}_{sc(\mathcal{R})} = (V, E)$*, a* joining tree *of tuples* $JT$ *is a tree of tuples where each edge* $(t_i, t_j)$ *in* $JT$*, with* $t_i \in R_i$ *and* $t_j \in R_j$ *satisfies two properties: (1)* $e = (R_i, R_j) \in E$*, and (2)* $(t_i, t_j) \in R_i \bowtie R_j$*. The set of tuples of* $JT$ *is denoted by* $Tuples(JT)$*.*

Join consistent and connected tuple sets are joining trees of tuples that do not contain more than one tuple from the same table. These are the building block components of the full disjunction.

**Definition 2** (**Join consistent and connected tuple set**). *Given a database* $\mathcal{R} = \{R_1, ..., R_n\}$ *with schema graph* $\mathcal{G}_{sc(\mathcal{R})}$*, a* tuple set *of* $\mathcal{R}$ *is any set of tuples* $T = \{t_1, ..., t_m\}$ *consisting of at most one tuple from each relation (hence* $m \leq n$*).*

*We say that* $T$ *is* join consistent and connected *if there exists a joining tree* $JT$ *such that* $Tuples(JT) = T$*, i.e., the set of tuples of* $JT$ *coincides with* $T$*. We denote as* $JCC(T)$ *a set of tuples* $T$ *is join consistent and connected.*

**Example 2:** With reference to the database shown in Figure 1(a) with schema graph in Figure 1(b), the tuple set $T_0$=$\{c_0, s_0, m_0\}$ is join consistent: in this case there are 3 joining trees whose nodes coincide with $T_0$: $\{(c_0, s_0), (s_0, m_0)\}$, $\{(c_0, s_0), (c_0, m_0)\}$ and $\{(m_0, s_0), (c_0, m_0)\}$; $T_1$=$\{c_1, s_1, m_1\}$ is also join consistent: in this case there exists only the joining tree $\{(c_1, s_1), (s_1, m_1)\}$ whose nodes coincide with $T_1$; on the other hand, $T_2$=$\{c_2, s_1, m_1\}$ is not join consistent since $(c_2, s_1) \notin$ C$\bowtie$S.

**Definition 3** (**Full Disjunction**). *Let* $\mathcal{R}$ *be a set of relations. The* full disjunction *of* $\mathcal{R}$*, denoted* $FD(\mathcal{R})$*, is the set of all tuple sets* $T$ *of* $\mathcal{R}$*, such that (1)* $T \in JCC(T)$*, and (2)* $T$ *is maximal, that is, there is no join consistent and connected tuple set of* $\mathcal{R}$ *that properly contains* $T$*.*

The full disjunction is a set of tuple sets, having each item the same schema, regardless the tuple sets it involves. Let us consider a *join consistent and connected* tuple set $T$ of $\mathcal{R}$, and denote $embed_{\mathcal{R}}(T)$ the tuple that is obtained by firstly *joining* tuples of $T$ and then adding columns with null values for the remaining attributes of $sc(\mathcal{R})$. More formally, $embed_{\mathcal{R}}(T)$ is the tuple $t$ over $sc(\mathcal{R})$, such that for all attributes $A$ of $sc(\mathcal{R})$, if $T$ contains a tuple $t'$ with the attribute $A$, then $t[A] = t'[A]$; otherwise, $t[A] = \perp$. $embed_{\mathcal{R}}(t)$ is a shorthand notation for $embed_{\mathcal{R}}(\{t\})$.

**Definition 4** (**Full Disjunction Relation**). *The* full disjunction relation *of* $\mathcal{R}$*, denoted* $FDR(\mathcal{R})$*, is the relation on* $sc(\mathcal{R})$ *obtained as* $\{embed_{\mathcal{R}}(T) \mid T \in FD(\mathcal{R})\}$.

**Example 3:** Table 1(c) shows the full disjunction of the running example database, i.e. the possible "combinations" of the data in the original table according to the foreign key relationships. Note that only the identifiers of the tuples in the original tables are provided. Table 1(d) reports the full disjunction relation, where the tuple identifier used in Table 1(c) are substituted with the real tuples or null values if missing.

In the following, we will adopt the term full disjunction to refer also to its transformation in full disjunction relation.

## 3. The PARAFD Approach

The PARAFD process is based on the idea that the full disjunction of a set of relations $\mathcal{R}$ is obtainable as the union of the full disjunctions of all possible *spanning trees*[4] of its schema graph $\mathcal{G}_{sc(\mathcal{R})}$. This result, which is formally demonstrated in Section 3.1, allows us (1) to compute the full disjunction through the simple application of the full outerjoin operator; and (2) to split the computation process in a number of steps that can be executed in parallel (see Section 3.2).

### *3.1. Computing a Full Disjunction through Spanning Trees*

In this section we show that the full disjunction of a set of relations $\mathcal{R}$ with schema graph $\mathcal{G}_{sc(\mathcal{R})}$ can be obtained by the combination of the full disjunctions computed for each possible *spanning tree* of $\mathcal{G}_{sc(\mathcal{R})}$, after the removal of the tuple sets that have been already generated by other spanning trees or are contained in other tuple sets. Moreover, we show that the full disjunction of a spanning tree can be computed by means of the full outerjoin operator.

To implement this procedure, we need to extend the full disjunction definition to be applied to schema subgraphs.

**Definition 5** (**Full Disjunction of a Schema Subgraph**). *Let $\mathcal{R}$ be a set of relations with schema graph $\mathcal{G}_{sc(\mathcal{R})}$. Given a connected subgraph $SG = (V_{SG}, E_{SG})$ of $\mathcal{G}_{sc(\mathcal{R})}$, the full disjunction of the set of relations $V_{SG}$ is called full disjunction of $SG$ and denoted by $FD(SG)$.*

The *subsumption operator* [2] allows us to remove duplicated and contained tuple sets.

**Definition 6** (**Subsumption**). *Given two tuple sets $T'$ and $T$, we say that $T'$ subsumes $T$ if and only if $T' \supseteq T$. The unary* subsumption operator $\downarrow$ *denotes the removal of subsumed tuple sets from a set of tuple sets $\mathcal{X}$:*

$$\downarrow\mathcal{X} \quad = \{T \in \mathcal{X} \mid \nexists T' \in \mathcal{X} : T' \supseteq T\} \tag{1}$$

**Example 4:** In the example of Figure 1, we can build the following three spanning trees: $ST_1 = (\mathcal{R}, \{e_1, e_2\})$, $ST_2 = (\mathcal{R}, \{e_2, e_3\})$, $ST_3 = (\mathcal{R}, \{e_1, e_3\})$. It is easy to verify that $\{c_0, s_0, m_0\}$ is in both $FD(ST_1)$ and $FD(ST_2)$, while $\{c_1, s_1, m_1\} \in FD(ST_1)$ but $\{c_1, s_1, m_1\} \notin FD(ST_2)$ (since $(c_1, m_1) \notin \texttt{C} \bowtie \texttt{M}$). Moreover, $FD(ST_2)$ contains the tuple set $\{c_1, s_1\} \in FD(ST_2)$ which is subsumed by $\{c_1, s_1, m_1\}$. Then, to obtain $FD(\mathcal{R})$ starting from the full disjunction of its spanning trees we need to eliminate such subsumed tuple sets as stated by the following theorem.

Based on Definitions 5 and 6, Theorem 1 demonstrates how we can compute the full disjunction of a set of relations by means of the spanning trees computed on its schema

[4] A spanning tree $ST$ of $\mathcal{G}_{sc(\mathcal{R})}$ is a subgraph of $\mathcal{G}_{sc(\mathcal{R})}$, which has all vertices covered with minimum possible number of edges.

graph.

**Theorem 1:** *Given a database $\mathcal{R} = \{R_1, ..., R_n\}$ with schema graph $\mathcal{G}_{sc(\mathcal{R})}$, and $\mathcal{ST}$ the set of all spanning trees of $\mathcal{G}_{sc(\mathcal{R})}$, the full disjunction $FD(\mathcal{R})$ can be obtained as:*

$$FD(\mathcal{R}) = \Big\downarrow \bigcup_{ST \in \mathcal{ST}} FD(ST) \tag{2}$$

*Proof.* The proof proceeds by demonstrating that the right-hand side (RHS) of Equation 2 includes only tuples satisfying both the properties required to be a full disjunction for $\mathcal{R}$ (see Definition 3). Then we demonstrate that it is not possible that a tuple set which is part of the full disjunction of $\mathcal{R}$ is not contained in RHS of Equation 2.

A tuple set $T \in RHS$ of Equation 2 is a join consistent and connected tuple set. First of all, $T$ is an element of a full disjunction. Then, by definition, T is a join consistent and connected tuple set. Moreover, each tuple set $T$ is maximal since the subsumption operator removes contained tuple sets.

Finally, we prove by reductio ad absurdum that it cannot exist a tuple set $T \in FD(\mathcal{R})$ such that $T \notin RHS$. But if $T \in FD(\mathcal{R})$, there exists by construction at least a spanning tree $ST$ of $\mathcal{G}_{sc}(\mathcal{R})$ such that $T \in FD(ST)$. But $ST$ should be in $\mathcal{ST}$, otherwise there would exist a spanning trees of $\mathcal{G}_{sc(\mathcal{R})}$ which is not part of the set of all spanning tree of $\mathcal{G}_{sc(\mathcal{R})}$.

□

In the rest of this subsection we synthesize the technique for computing the full disjunction of a tree, introduced in [5]. Let us use $R_1 \mathbb{K} R_2$ to denote the full outerjoin of $R_1$ and $R_2$. Full outerjoin is not associative: different execution orders generate different results. Left-deep outerjoins allow us to introduce a specific order.

**Definition 7** (**Left-deep Outerjoin**). *The left-deep outerjoin of $(R_1, \ldots, R_n)$, denoted by $\mathbb{K}(R_1, \ldots, R_n)$, is defined as follows: $\mathbb{K}(R_1, R_2) = (R_1 \mathbb{K} R_2)$, and, recursively, for $n > 2$, $\mathbb{K}(R_1, R_2, R_3, \ldots, R_n) = \mathbb{K}(R_1 \mathbb{K} R_2, R_3, \ldots, R_n)$.*

Given a connected set of relations $\mathcal{R} = \{R_1, ..., R_n\}$, a *connected-prefix ordering* of $\mathcal{R}$ is an ordering $R_1, ..., R_n$ such that $\{R_1, ..., R_i\}$ is connected for all $1 \leq i \leq n$. The Proposition 3.1. given in [5] shows that when the scheme graph is a tree, a connected-prefix ordering yields a left-deep outerjoin that is equivalent to the full disjunction. We can apply this result to a spanning tree: given $\mathcal{R} = \{R_1, ..., R_n\}$, let $ST = (V, E)$ be a spanning tree of $\mathcal{R}$. If $R_1, ..., R_n$ is a connected-prefix ordering of $ST$, then $FD(ST) = \mathbb{K}(R_1, \ldots, R_n)$.

**Example 5:** With reference to the spanning tree $ST_1 = (\mathcal{R}, \{e_1, e_2\})$, where $e_1 = (\mathtt{M}, \mathtt{S})$ and $e_2 = (\mathtt{C}, \mathtt{S})$ the full disjunction relation $FD(ST_1)$ can be obtained by computing one of the following two outerjoin sequences: $(\mathtt{C} \mathbb{K} \mathtt{S}) \mathbb{K} \mathtt{M}$ and $(\mathtt{M} \mathbb{K} \mathtt{S}) \mathbb{K} \mathtt{C}$.

### *3.2. Full Disjunction of a Spanning Tree: Computation by Hash Star Join*

We showed in the previous section that the computation of full disjunction of a set of relations $\mathcal{R}$ with schema graph $\mathcal{G}_{sc(\mathcal{R})}$ can be decomposed in a number of computations,

one for each spanning tree we can build over the schema graph $\mathcal{G}_{sc(\mathcal{R})}$. We then showed that we can use a full outerjoin sequence to compute the full disjunction of a spanning tree.

In this section, we introduce an efficient way to compute full outerjoin sequences that are based on a novel parallel algorithm for performing hash star joins and that is described in Algorithm 3. This algorithm can efficiently perform multi-relation joins, by reducing the communication costs.

To be able to apply the hash star join, we introduce Theorem 2, where we show how to model the spanning trees as sets of *star trees*. Given the schema graph $\mathcal{G}_{sc(\mathcal{R})}$, let $TR$ a tree of $\mathcal{G}_{sc(\mathcal{R})}$. Let $TR^\star$ be the *star subtree* obtained as a subgraph of $TR$ by considering as center vertex the vertex of $TR$ with maximum degree[5]: $TR^\star = (R^c, \{R_1^s, \ldots, R_n^s\})$, where $R^c$ is the *center* vertex and $R_i^s, \forall i = 1, \ldots, n$, are the adjacent *satellite* vertices.

**Theorem 2:** *Given the schema graph $\mathcal{G}_{sc(\mathcal{R})}$, let $TR$ be a tree of $\mathcal{G}_{sc(\mathcal{R})}$. Then:*

$$FD(TR) = \begin{cases} FD(TR^\star) & if \;\; TR \;\; coincides \;\; with \;\; TR^\star \\ FD(\Gamma(TR, TR^\star)) & otherwise \end{cases} \tag{3}$$

*where $\Gamma(TR, TR^\star)$ is the tree defined as follows*

1. *removing the star subtree $TR^\star$ from $TR$*
2. *adding the relation $R^\star = FD(TR^\star)$ to $TR$ and, for each edge $(R_i^s, R) \in TR$ involving a satellite $R_i^s$, adding a corresponding edge $(R^\star, R)$ to $TR$.*

*Proof.* The proof is based on the fact that Equation 3 generates a *connected-prefix ordering* of the vertices of $TR$; therefore, by applying the aforementioned Proposition 3.1 given in [5], the full disjunction can be computed as a left-deep outerjoin sequence. First of all, for a star tree $TR^\star = (R^c, \{R_1^s, \ldots, R_n^s\})$, it is trivial to prove that $R^c, R_1^s, \ldots, R_n^s$ is a connected-prefix ordering of $\{R^c, R_1^s, \ldots, R_n^s\}$, for any ordering of *satellite* vertices $\{R_1^s, \ldots, R_n^s\}$. Then, by iteratively applying the *tree contraction* defined by the $\Gamma()$ function, we obtain a connected-prefix ordering of $TR$.

Let us show this result intuitively with the example in Figure 2, where we consider, on the left, a tree $TR_1$ with 12 vertices denoted by $0, 1, \ldots, 11$. $TR_1^\star$ is the star subtree of $TR_1$ with center vertex 0 and satellite vertices $\{1, 2, 3, 4, 5\}$, then we obtain $TR_2 = \Gamma(TR_1, TR_1^\star)$, where $R_1^\star = FD(TR_1^\star)$. $TR_2^\star$ is the star subtree with center vertex $R_1^\star$ and satellite vertices $\{6, 8, 10\}$, then we obtain $TR_3 = \Gamma(TR_2, TR_2^\star)$, where $R_2^\star = FD(TR_2^\star)$. $TR_3^\star$ is the star subtree with center vertex $R_2^\star$ and satellite vertices $\{7, 9, 11\}$: since $TR_3^\star = TR_3$ the process stops. It is trivial to prove that $0, 1, 2, 3, 4, 5$ is a connected-prefix ordering of the vertices of $TR_1^\star$. It is also trivial to prove that $0, 1, 2, 3, 4, 5, 6, 8, 10$ is a connected-prefix ordering of the union of the vertices of $TR_1^\star$ and $TR_2^\star$, since, for each edge $(R_i^s, R) \in TR_1$ involving a satellite $R_i^s$, of $TR_1^\star$, a corresponding edge $(R_1^\star, R)$ to $TR_2^\star$ is added. In the same way, $0, 1, 2, 3, 4, 5, 6, 8, 10, 7, 9, 11$ is a connected-prefix ordering of the union of the vertices of $TR_1^\star$, $TR_2^\star$ and $TR_3^\star$, i.e., of the vertices of $TR_1$. □

[5] Random choice in case of a tie.

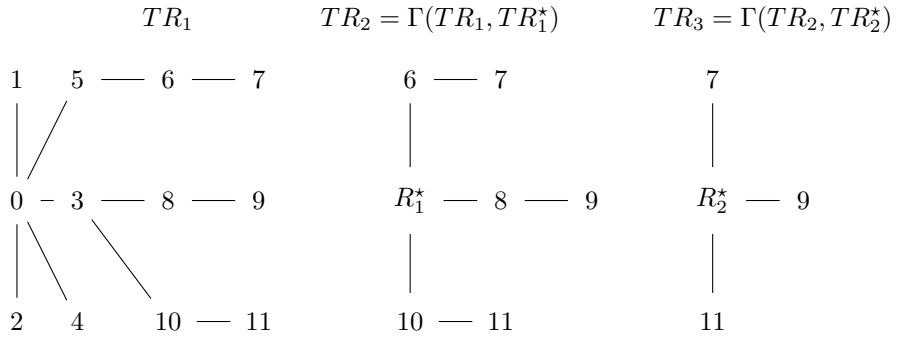



Figure 2: Example of application of the Theorem 2

**Algorithm 1:** `ComputeFD` - Full Disjunction of a database

**Input :** A database $\mathcal{R}$ with schema graph $G$
**Output:** The full disjunction of $\mathcal{R}$

1 $\mathcal{X} \leftarrow \emptyset$;
2 $\mathcal{ST} \leftarrow GetSpanningTrees(G)$;
3 **foreach** $ST \in \mathcal{ST}$ **do**
4 $\quad \mathcal{X} \leftarrow \mathcal{X} \cup ComputeFD\_ST(ST)$;
5 $FD \leftarrow SubsumptionOperator(\mathcal{X})$;
6 **return** $FD$;

**Example 6:** The big picture of the PARAFD approach is shown in Figure 3: the spanning trees of a database schema graph are computed. We obtain the full disjunction of each spanning tree through the application of Hash Star Joins. The full disjunction of the database is generated by collecting the results obtained for each spanning tree and removing subsumed elements.

### *3.3.* PARAFD *implementation*

Algorithm1 shows the implementation of PARAFD. It takes as input a database $\mathcal{R}$ with schema graph $G$ and it computes its full disjunction. First of all, the set $\mathcal{X}$ that will contain the resulting full disjunction (line 1) is initialized. Then, it computes all spanning trees from the input schema graph by means of the *GetSpanningTrees* function (line 2), which is described in Section 3.3.1. The full disjunction of each spanning tree is then computed (lines 3-4). This computation is performed by the function *ComputeTS* which is described in Section 3.3.2. At the end of the iteration, $\mathcal{X}$ will contain the full disjunction computed for each spanning tree (line 4). The full disjunction $FD$ of $\mathcal{R}$ is finally generated by removing duplicated and subsumed tuple sets from $TS$ (line 5) and is returned as output (line 6). A description of the tuple set subsumption process is provided in Section 3.3.3.

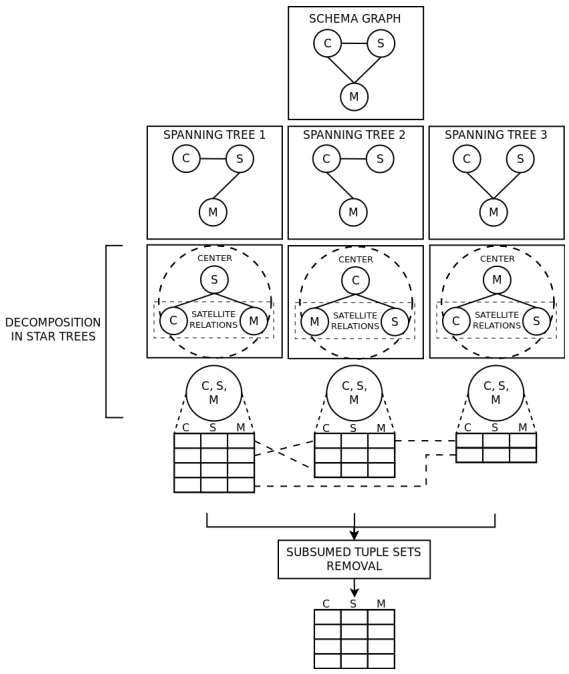



Figure 3: PARAFD applied to the running example

*3.3.1. Computation of spanning trees over a schema graph*

The computation of all spanning trees of a graph is a problem that has been already addressed in the literature. Among the existing solutions, we adopted the algorithm presented in [9], based on backtracking and depth-first search, which runs in $O(V+E+VN)$ time, where $V$, $E$, and $N$ are the number of vertices, edges, and spanning trees, respectively. Since in the real databases the number of tables and foreign keys relationships is limited, generating and computing the spanning trees is a feasible approach.

*3.3.2. Computation of the full disjunction of a spanning tree*

**Algorithm 2:** `ComputeFD_ST` - Full Disjunction of a Spanning Tree

**Input :** A spanning tree $ST$.
**Output:** The full disjunctions of $ST$.

1 **do**
2 $\quad X \leftarrow StarGraph(ST)$;
3 $\quad RX \leftarrow HashStarJoin(X)$;
4 $\quad ST \leftarrow \Gamma(ST, RX)$;
5 **while** *$ST$ is not a single vertex*;
6 **return** $RX$;

Algorithm 2 shows the implementation of the function `ComputeFD_ST` for computing the full disjunction $FD$ generated by a spanning tree $ST$, on the basis of Equation 3. This process starts with the identification of a *star graph* $X$ in $ST$ (line 2). Then, the

procedure *HashStarJoin*, described in Algorithm 3, computes the full disjunction of $X$ (line 3). Line 4 builds the new spanning tree to be evaluated by means of the $\Gamma$ function defined in Theorem 2. The process iterates until $ST$ is constituted of a single vertex only (line 5). Finally, the full disjunction is returned (line 6).

Performing efficiently a join operation in a distributed environment is usually a critical task. Communication costs represent the main bottlenecks because the computation time is usually less expensive than the time required for data distribution/shuffling. The Hash Star Join technique is similar to *SHJ* proposed in [10] (see related work section) for joining in a parallel/cluster architecture the fact table of a data warehouse system with its corresponding dimensions. The distribution of the data to be joined is done by applying a *shipping function* that decides the host cluster of each record in a table. The shipping function is applied to a column (partitioning key) of the table to partition. When performing a join operation, if the joining key is the same as the partitioning key used for both input tables, then the join can be locally executed within each cluster. In the other case, the join operations need for a re-partitioning of the data.

The strategy adopted for partitioning the tables is crucial for obtaining high time performance computations. In our Hash Star Join approach, we would like to exploit the fact that adjacent vertices in a star tree can share the same joining key with the center vertex and thus all the related joins can be performed locally within a cluster. Based on this consideration, the idea is to analyze the join associations existing between the input tables and perform their partitioning based on the most frequent join attributes. This lead to a specific execution order of the join operations that maximizes the tables involved in each operation, and reduces the amount of data to transmit over the network.

The principle behind the standard hash join is to limit the number of total comparisons: only the tuples that fall in the same bucket are checked if they are joined consistent. In our implementation the main changes made to this basic logic are: (a) creation of a distributed hash table, that allows us to parallelize for each bucket the join computation among different processes (b) involvement of an arbitrary number of tables and (c) application of the outerjoin operator within a bucket.

**Algorithm 3:** `HashStarJoin`

**Input :** A star tree $ST$ with center in relation $R_k$ and adjacent relations $Adj_{R_k}$.
**Output:** A set of tuple sets representing the full disjunction $FD^*$ of $X$.

1 $FD^* \leftarrow \emptyset$;
2 $SubStars \leftarrow ClusterByFK(Adj_{R_k})$;
3 **foreach** $subStar \in SubStars$ **do**
4 $\quad P \leftarrow PartitionTablesByFK(subStar)$;
5 $\quad$ **map** $p \in P$
6 $\quad\quad R(p) \leftarrow LeftDeepOuterJoin(subStar(p))$
7 $\quad FD \leftarrow \bigcup_p R(p)$;
8 $\quad FD^* \leftarrow FD^* \,\overline{\bowtie}\, FD$;
9 **return** $FD^*$;

Algorithm 3 shows the procedure for generating the sets of tuple sets from a star graph centered in $R_k$ and with satellite relations $Adj_{R_k}$. We start initializing the set $FD^*$ that will contain the resulting FD (line 1). In line 2, given a star tree $ST$ in input, the function $ClusterByFK$ finds a sequence of subStars $SS_1, ...SS_n$ such that, for each

$SS_i$, the vertices share the same joining key, and $SS_i$ has more vertices than $SS_{i+1}$.

**Example 7:** If we consider the star tree $ST_2$ (see Example 4) with $City$ as center vertex, the sequence is just $SS_1 = ST_2$. The reason is that both the adjacent vertices $Membership$ and $State$ share the City joining key. If we consider the star tree $ST_3$ with $Membership$ as center vertex, the sequence is $SS_1 = \{M, C\}$ *and* $SS_2 = \{S, C\}$ as the there is no a common join attribute.

Then, the algorithm iterates over the obtained sequence of $subStars$ (lines 3-8). At each iteration a set $Substar$ is extracted from $SubStars$. This set of tables is then partitioned through a hash function that is applied to the most common attribute in their join associations. According to this partition schema, we distribute the tuples to different nodes. Tuple sets are thus computed separately by each node through a MapReduce process[6] (lines 5-6). For each partition the $LeftDeepOuterjoin$ operator joins the tuples following a left outerjoin sequence. When all nodes have completed the computation, the results are collected and stored in $FD$ (line 7). This operation terminates the elaboration of the current cluster of relations. Finally, the tuple sets computed from the current cluster are merged with the results of the previous clusters via the full outerjoin operator (line 8). All tuple sets are then returned as output (line 9).

In this perspective, computing the full disjunction for each spanning tree in isolation, as done by Algorithm 2, can be computationally very expensive. Spanning trees can differ with each other by few edges, and running several times join operations on the same tree portions results in a number of overlapping full disjunctions and an unjustified increase in the overall execution time of the algorithm. To address this issue, we implemented a mechanism for storing the tuple sets generated by sequence of edges of the spanning trees to be able to retrieve and reuse them if the same tree portion is navigated in another computation.

**Example 8:** Consider the tree shown in Figure 4(b), where the CITY node has STATE, MEMBERSHIP as adjacent nodes. The join attributes between each pair of tables to be combined by the full outerjoin operator are a) $S.capital \rightarrow C.name$ belonging to the tables STATE and CITY; b) $M.cityorg \rightarrow C.name$ belonging to the tables MEMBERSHIP and CITY . $C.name$ is an attribute common to both the join associations. This information is exploited to perform a single partition of the three tables considered.

*3.3.3. Generating the full disjunction*

Our technique for removing subsumed tuple sets relies on a *set-trie* data structure [11] to store the full disjunction produced by each spanning tree. A set-trie is an extension of a trie [12] which, in addition to simply verifying the membership of an element within it, supports search operations on subsets and supersets. In particular, set-tries use prefixes of common elements to index the elements thus enabling the efficient identification of their subsets / supersets. The complexity of these operations is $O(c * |set|)$, where $|set|$ represents the size of the input, and $c$ is a constant.

[6]Note that the *map* operator in Algorithm 3 represents the map method in a mapreduce programming model. The operations performed by the map will be executed in parallel by a mapper for each data partition. For sake of simplicity we omitted to explicit in the algorithm the reduce process since it has been implemented with an identity function.

**Algorithm 4:** `RemoveSubsumed`

**Input :** A set of tuple sets $TS$.
**Output:** A set of tuple sets $TS^{\star}$, with no subsumed items.

```
1  P ← Coverage(TS);
2  TS* ← ∅;
3  map p ∈ P
4      Q_max ← ∅;
5      for T ∈ p do
6          Q_max.add(T);
7      Trie^p_SET ← ∅;
8      while Q_max.isNotEmpty() do
9          T ← Q_max.pop();
10         if T ⊈ T', T' ∈ Trie^p_SET then
11             Trie^p_SET.add(T);
12 TS* ← ⋃_p Trie^p_SET;
13 return TS*;
```

Algorithm 4 shows the functionality implemented for removing subsumed tuple sets. In line 1 the *Coverage* function is applied for computing a specific *covering set* $\mathcal{C}$ of $TS$ (i.e., a collection of subsets of $TS$ whose union is $TS$) such that, for each tuple $t$ of $TS$, there is a (unique) item of $\mathcal{C}$ containing all and only the tuple sets with $t$. This operation can be easily done with a MapReduce implementation. Then, in parallel for each partition[7] (lines 3-11), the constituting tuple sets are extracted and sorted by size in descending order in the $Q_{max}$ priority queue (lines 4-6). The tuple sets are progressively indexed in a set-trie (line 7) for being analyzed as possible results. Lines 8-11 exploit the set-trie to verify if tuple sets are subsumed. If the tuple set is not contained in any previous full disjunction (line 10), it is inserted in the set-trie (line 11). Finally, full disjunctions computed in each partition are merged and returned (lines 12-13).

### *3.4. Approximating the full disjunction*

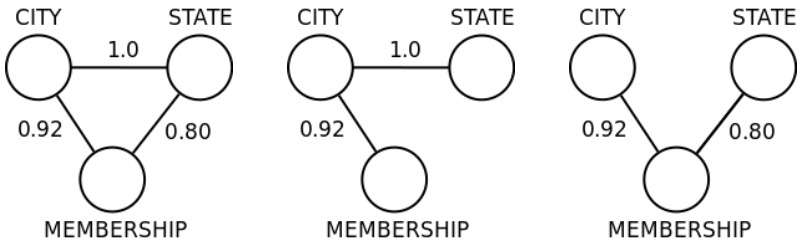
There are scenarios where the computation of the complete set of full disjunctions is not needed. In these situations, approximating the complete set of tuples in a full disjunction with the most "meaningful" ones decreases the computation time with some loss of information.

The selection of an approximate set of full disjunction is based on the analysis of the spanning trees. In particular, we adopted the Pointwise Mutual Information as the measure for weighting the edges of a graph and implemented a well-known Algorithm ([13]) for computing the top-k maximum cost spanning trees.

Pointwise Mutual Information (PMI) has been largely applied in computer science and it provides a correlation measure between two entities, evaluating their probability of joint occurrence in the hypothesis of absence and presence of statistical dependence.

The database research community typically relies on PMI-based measures to weight the cohesion of tables connected via foreign keys in a database [14, 15].

[7] As in Algorithm 3 the reduce process is not represented since implemented through an identity function




(a) Weighted schema graph

(b) Maximum spanning tree

(c) Minimum spanning tree

Figure 4: Running example schema graph and some derived spanning trees

**Example 9:** In Figure 4(a) the edges of the schema graph representing our running example are weighted according to PMI values. Note that the values show high cohesion between the tables STATE - CITY and a lower cohesion between the couple of tables CITY - MEMBERSHIP and STATE - MEMBERSHIP. The maximum and minimum spanning trees are shown in Figures 4(b) and 4(c).

## 4. Reference techniques for computing the full disjunction

This section provides an overview of two main techniques for computing full disjunctions proposed in the literature: $IncrementalFD$ [4] and $BiComNLOJ$ [5]. These techniques have been extended in this paper to perform with equijoins (in the original papers they have been designed to work with natural joins) and with parallel computing techniques (4 variations of $IncrementalFD$ have been developed). These techniques are used in Section 6 as a baseline for evaluating PARAFD.

### *4.1. IncrementalFD*

$IncrementalFD$ [4] performs an incremental computation of full disjunction. It iterates over all the tables and, for each of them, it computes a number of tuple sets which are "candidate" to belong to the full disjunction: given a table $R$ in a set of relations $\mathcal{R}$, the candidate full disjunction for $R$ is the subset of $FD(\mathcal{R})$ that contains tuple sets with a tuple from $R$. Candidate tuple sets do not contain subsumed items (i.e., tuple sets are already maximal), then their simple union generates the resulting $FD(\mathcal{R})$.

The candidate full disjunctions are computed by means of two processes: the *extension* operation that takes a connected and join consistent tuple set and adds a series of tuples from the tables which have not been already used, thus creating another connected and join consistent tuple set, and the *variation* operation that generates connected and join consistent tuple sets by substituting a tuple in a connected and join consistent tuple set with another tuple from one of the tables that have already been examined.

Algorithm 5 provides more details on $IncrementalFD$. In line 1 $FD$, the set that will contain the resulting full disjunctions, is initialized. Line 2 shows that the process iterates over all the tables. For each table $R_i$, $Incomplete$, the variable that stores the tuple sets under development, is initialized. Then, in line 6, the Algorithm iteratively extracts a tuple set T from $Incomplete$ and applies the extension and variation processes.

**Algorithm 5:** IncrementalFD (adapted from [4])

**Input :** A database $\mathcal{R}$.
**Output:** The full disjunction of $\mathcal{R}$.

```
1  FD ← ∅;
2  for R_i ∈ R = {R_1, ..., R_n} do
3      Incomplete ← ∅;
4      for t_ij ∈ R_i do
5          Incomplete.add({t_ij});
6      for T ∈ Incomplete do
7          if T is not maximal then
8              T ← EXTENSION(T);
9          VAR(T) ← VARIATION(T);
10         for T' ∈ VAR(T) do
11             if T' contains a tuple from R_i then
12                 if T' ∉ TS then
13                     Incomplete.add(T');
14                     if T' is superset of an S ∈ Incomplete then
15                         Incomplete.remove(S);
16         FD ← FD ∪ T;
17 return TS;
```

The **extension process** consists in adding to the current tuple set T a series of tuples belonging to database tables which have not been already used to form $T$ (otherwise Definition 2 would be violated). The resulting tuple set has to be connected and join consistent.

The **variation process** aims to discover other tuple sets involving the same tables as the current $T$. To do it, a scan on all database tables (excepting $R_i$) is performed to find possible tuples to substitute with the ones in the current tuple set, and generating new connected and join consistent tuple set.

Each tuple set obtained by the variation process is evaluated to be a possible item in the full disjunction. This happens when (lines 11-13) they a) contain a tuple belonging to $R_i$, b) have not already been generated and c) do not correspond to a connected and join consistent super-set of an already generated full disjunction. In this latter case, a replacement of the low cardinality tuple set with $T'$ is needed (lines 14-15).

Once an item has been submitted to the variation task, its elaboration is considered as completed, and it is added to the final results of the algorithm.

**Example 10:** Figure 5 shows a simple example of the application of the Algorithm to the running example. The algorithm starts the iteration with the STATE table by inserting all its tuples in the *Incomplete* variable (lines 2-5). The maximal extension of the first tuple $s_0$ is computed and a variation process is applied to the resulting tuple set (lines 6-9, see Step 2 in the Figure). The algorithm proceeds by evaluating if the variation can be considered as a full disjunction (lines 10-15). A similar extension and variation process is applied to the other tuples $s_1$ and $s_2$ as shown in the Figure. Notice that the extension $\{s_2,m_3\}$ is maximal even if it does not include any tuple of the CITY table. Nevertheless, it will not generate any variation (see the empty set).

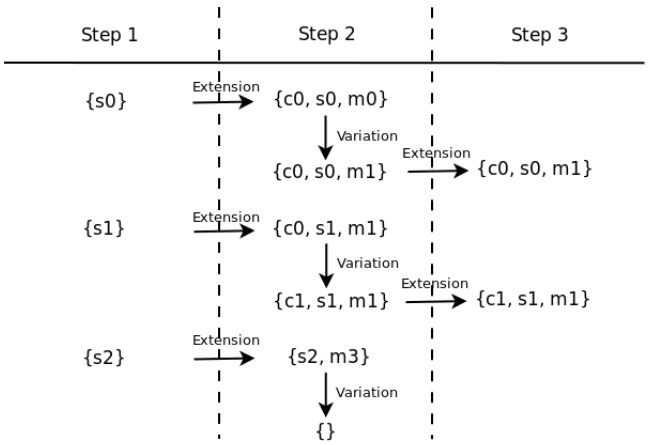



Figure 5: Candidate full disjunction of State

*4.2. The BiComNLOJ approach*

*BiComNLOJ* [5] is an approach for the full disjunction computation that exploits the concept of polynomial delay. This concept requires that the time interval between the production of two successive solutions varies in a polynomial manner with respect to the size of the input data, thus assuring high performance. In particular, *BiComNLOJ* consists of two main components: one for the calculation of sequences of left deep outer-joins in an acyclic graph (*NestedLoopOuterJoin* - *NLOJ*) and one for the calculation of full disjunctions in a general graph which has a quadratic delay (PDelayFD). These components were initially assembled to compute the full disjunctions according to the following steps:

1. calculation of the biconnected components of the schema graph
2. calculation of the full disjunctions for each biconnected component using the PDelayFD algorithm
3. combination of the full disjunctions deriving from each biconnected component through the execution of the *NLOJ* algorithm respecting a specific order

This computational schema however does not guarantee a polynomial delay execution as the time needed to produce the full disjunction for a single biconnected component is exponential. In order to achieve the polynomial delay property the combination of the intermediate full disjunction produced by the different biconnected components, through the *NLOJ* algorithm, is progressively executed. Firstly, the full disjunction deriving from the first biconnected component is computed. Then, starting from these results, the full disjunction items of the second biconnected component are then computed, and so on. In order to guarantee the correctness of the combination process a specific order must be used: this is called "strong connected-prefix order", and it imposes that two successive biconnected components have to be connected by a connecting relation. This is a relation that either appears in both the biconnected components or is directly connected, through a join condition, with a relation of the previous biconnected component. This new

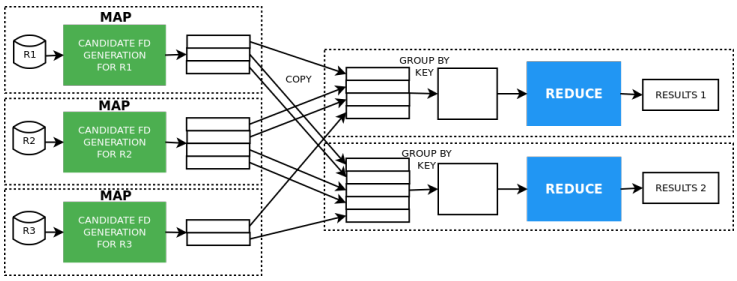



Figure 6: Table-driven naive parallelization

execution flow provides a method for full disjunction computation that is compliant with the polynomial delay property.

### *4.3. Parallelizing the IncrementalFD algorithm*

The logic of the $IncrementalFD$ algorithm can be easily parallelized as it consists of several independent computations operating on data of significant dimensions (e.g., multiple database tables). This section introduces four approaches to parallelize $IncrementalFD$ by means of a map reduce strategy.

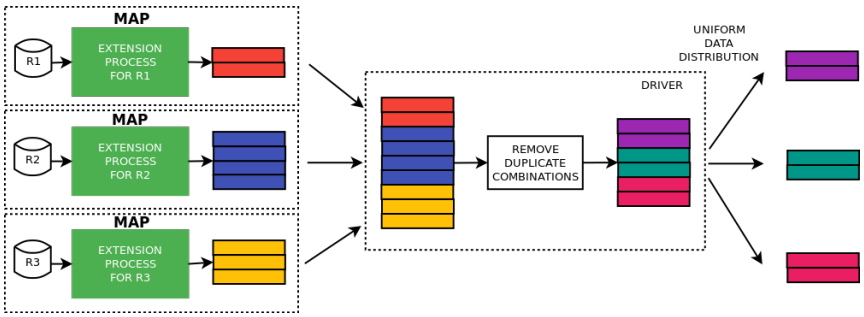
#### *4.3.1. Table-driven naive parallelization*

This approach is a direct variant of the $IncrementalFD$ algorithm, where the first iteration over the database tables (see line 2 of Algorithm 5) is executed in parallel. In this way, the approach simultaneously performs the creation of "candidate" full disjunction relations starting from the different tables. The parallelism adopted here is mainly applied at a "code level": the algorithm does not change, but it is the flow of execution that has been altered by inserting multiple workers operating simultaneously to perform parallel "candidate" full disjunction generation. Once the generation phase is completed, a deduplication task is performed to remove repeated tuple sets. The whole approach and the deduplication task can be easily implemented adapting the $IncrementalFD$ algorithm implementation into a MapReduce architecture, where the map task consists in a "candidate" full disjunction discovery process (there is a mapper for each table) and the reduce task directly removes the duplicated tuple sets in order to produce full disjunction.

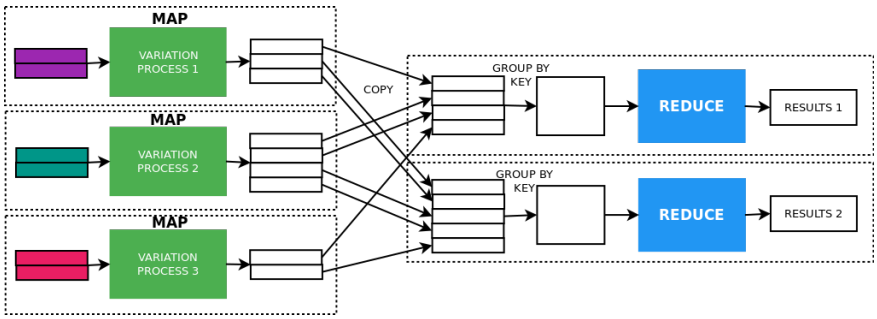
**Example 11:** Figure 6 exemplifies our approach. For each table, a map task performing the extension and the variation processes is created. In this way, all tuple sets originating from that table are computed. Then, a reducer task responsible for the removal of duplicated items is executed. The remaining tuple sets compose the full disjunction.

#### *4.3.2. Two-phase computation*

The idea behind this variation is to divide the process for generating the full disjunction into two phases parallelized via a MapReduce implementation as shown in Figures 7(a) and 7(b). The first phase implements the functionalities introduced in lines 2-8 of Algorithm 5. A mapper is initialized for each relation, with the goal of generating the first set of tuple sets. The subsequent reducers remove the duplications. The second




(a) Full Disjunction Generation - First Phase: Extension Process




(b) Full Disjunction Generation - Second Phase: Variation Process

Figure 7: The MapReduce phases of Checkpoint-based version

phase implements the rest of Algorithm 5, mainly by applying the variation process to the tuple sets given as input and generating the rest of the full disjunctions. Moreover, the data are in the second phase uniformly distributed to the mappers to optimize the process execution.

This approach largely improves the previous table-driven naive parallelization. Firstly, the duplicated tuples sets generated by the first phase are removed before the variation process is applied, thus avoiding the creation of tuple sets to be later on removed. Then, the uniform redistribution of the initial data to the mappers in the second phase optimizes the performances by balancing the workload of the workers executing the variation process. Finally, the algorithm implements a parallel and progressive mechanism for generating the full disjunction. The tuple sets resulting from the first phase constitute a preliminary result that the user can exploit in advance.

**Example 12:** One of the advantages introduced with the subdivision of the execution flow into two steps is the ability of removing duplicated tuple sets early, at the end of the first MapReduce task, thus generating an initial set of full disjunction relations. Figure 8 shows the application of the deduplication process to tables State and City of the running example. The first map tasks apply the extension process to the initial tuples. Duplicates are removed by the first reduce tasks. Then, a second MapReduce cycle applies the variation process and removes the duplicates.

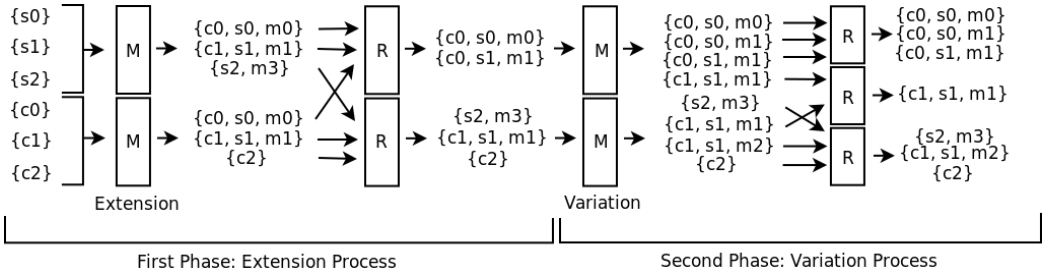



Figure 8: Two Phase Computation approach applied to the running example

| | Variation on {c0, s0, m0} | Variation on {c0, s1, m1} | Variation on {s2, m3} |
|---|---|---|---|
| c0 → | n.a. | n.a. | ~~{c0, s2, m3}~~ |
| c1 → | ~~{c1, s0, m0}~~ | **{c1, s1, m1}** | ~~{c1, s2, m3}~~ |
| c2 → | ~~{c2, s0, m0}~~ | ~~{c2, s1, m1}~~ | ~~{c2, s2, m3}~~ |
| m0 → | n.a. | ~~{c0, s1, m0}~~ | n.a. |
| m1 → | **{c0, s0, m1}** | n.a. | n.a. |
| m2 → | ~~{c0, s0, m2}~~ | ~~{c0, s1, m2}~~ | n.a. |
| m3 → | ~~{c0, s0, m3}~~ | ~~{c0, s1, m3}~~ | n.a. |

Figure 9: Block variation process applied to the running example

*4.3.3. Block-based parallelization*

The Block-based parallelization improves the performance of the approach by optimizing the execution of the variation process. Given a tuples set, the variation process replaces its composing tuples with other tuples belonging to the same tables. This task can be optimized if we are able to work on multiple tuple sets simultaneously. In this way, the cache usage is optimized and the total execution time decreases.

Therefore, the Block-based parallelization implements a change in the logical data unit processed by the algorithm, by grouping all tuples sets generated from the same tables in the previous extension block and computing the variation of the entire group.

**Example 13:** Let us consider the three tuple sets generated by the extension task on the tuples of the State table: (1) $\{c_0, s_0, m_0\}$, (2) $\{c_0, s_1, m_1\}$ and (3) $\{s_2, m_3\}$. The variation process will scan all tuples of the City and Membership tables to generate possible variations. Figure 9 shows the process and the results obtained, where the columns represent the input tuple sets and the row the possible variations. The Figure shows that only two valid results are generated: $\{c_0, s_0, m_1\}$ and $\{c_1, s_1, m_1\}$. The other cases generate tuple sets which are not connected and join consistent. Note that when the variation tuple is already included in the considered tuple set no variation operation is applied ("n.a." in Figure).

*4.3.4. Checkpoint-based version*

The block-based version can generate workload imbalance among the workers. To provide a partial solution to this issue, a checkpoint system has been implemented to periodically interrupt the execution of the workers, collect the results produced up to that

point and produce a new distribution of the data. The frequency of the re-balancing is established as proportional to the number of new full disjunction discovered with respect to the number of tuple sets taken in input by the considered worker. When the difference between these two dimensions exceeds a fixed threshold the checkpoint system is triggered.

## 5. Related Work

Section 4 has introduced the main existing techniques for computing the full disjunction of a relational database. In this section, we introduce other related work on the available technologies for supporting parallel computing, and some approaches for performing the join in distributed and parallel frameworks.

### *5.1. Technology supporting parallel computing*

Recently, a large number of technologies and paradigms have been developed to efficiently manage high data volumes with reduced costs. MapReduce [16] was a first product developed to address the emergence of the high scalability and flexibility requirements imposed by large amounts of data. Its main advantage was its ability to hide implementation details in a parallel environment through the adoption of a distributed programming paradigm based on two primitive functions: Map and Reduce. Its simplicity, flexibility and fault tolerance have immediately made it a tool of paramount importance for managing large amounts of data. Several implementations of its paradigm have been developed. Among them, the most used is Apache Hadoop[8]. However, this model has shown some limitations. The main significant ones are: a) the use of a non-declarative programming paradigm based on its two primitives only, b) the inability to exploit the advantages deriving from the structured organization of some information, i.e., all data have to be organized in a key-value form, and c) the adoption of a rigid dataflow articulated in fixed phases such as reading data from a distributed file system, applying a MapReduce job and storing the results in a distributed file system. Within an iterative logic, the repeated application of a sequence of MapReduce jobs involves, due to the lack of storage of an intermediate state, intense use of I/O operations that can easily degrade performance. In response to these needs, new approaches have been introduced. Apache Hive[9] and Apache Pig[10] were the first tools integrating the MapReduce paradigm into a declarative logic, similar to the SQL one. Several wrappers have also been introduced to use structured data with a schema instead of key-value pairs. Finally, several frameworks have been created to achieve high performance even in the presence of iterative and interactive data processing. Apache Spark[11] represents the major exponent of this category of frameworks. It provides the most complete, reliable and performance solution. Thanks to an in-memory computing model, in fact, it is up to 100 times faster than Apache Hadoop and supports structured data analysis based on a declarative logic.

[8] http://hadoop.apache.org/
[9] https://hive.apache.org/
[10] https://pig.apache.org/
[11] http://spark.apache.org/

*5.2. Join in a distributed environment*

The join operator is a useful tool to integrate information from different data sources. Its operation, however, does not fit well in a distributed scenario where the information to be integrated can be divided between different nodes and therefore its integration requires high network traffic.

Parallel join processing originates from the work on the early parallel database systems, such as Bubba [17], PRISMA/DB [18] and GAMMA [19], where hash-based partitioning was used to distribute the join argument to multiple machines in a cluster. Evaluating a multi-join query via hashing in parallel over a shared-nothing environment has also been investigated in the literature. Different parallel processing strategies such as left-deep and right-deep [20], segmented right-deep[21], zigzag tree [22] and other variations [23] have been proposed. However, most of them report their results based on simulations, while we report our results based on a working distributed system.

Within a MapReduce framework, the join operation can be performed in two different ways: Map-side join and Reduce-side join [24]. The first family of joins includes Map-Merge joins and Broadcast joins. According to the first approach, the tables to be integrated are already partitioned in the distributed file system on the join key and a merge phase is applied through Map functions. The broadcast join instead exploits the possibility of replicating and storing in the memory of each mapper the table of smaller size in order to carry out the comparisons of joins more efficiently. In the Reduce-side joins, the most common strategy is the repartition join [25], which labels, in the map phase, the tuples according to the provenance table, partitions the tuples based on the join key and performs the tuple comparison in the reduce phase. This corresponds to the application of a standard hash join in a distributed environment. Although other MapReduce join approaches have been proposed, such as Map-Reduce-Merge [26] and Map-Join-Reduce [27], many implementations provided by higher-level systems built above MapReduce, like the Hadoop-based systems and those integrated within Apache Spark, are available. However, these systems provide implementations that can work only with two tables [28]. Therefore, the application of an integration task by joining $n$ data sources, which represents the typical scenario with the full disjunction, is divided into several phases. Concurrent join [29] and Scatter-Gather-Merge [28] algorithms have been proposed to efficiently implement star joins.

The need to integrate $n$ tables represents also a typical scenario in the field of business intelligence, where star queries require that a central table, i.e., the fact table, is integrated with information extracted from other tables, i.e., the dimension tables. Most of star query implementations [30] exploit the different dimensionality of the tables involved (i.e., the fact table is typically greater than the dimension tables), therefore they do not provide a generic strategy to operate. [10] introduces a technique called star hash join (SHJ), which provides an optimization of a left-deep tree-shaped query plan applied in a datawarehouse scenario. SHJ solves a problem similar to our implementation, but applies a less efficient solution based on a sequence of joins. In particular, the dimension tables are partitioned by their primary keys, and the fact table is partitioned by one of its foreign keys. Hence, only one join between one dimension table and the fact table may be performed locally within each cluster.

## 6. Experimental evaluation

We conducted a large number of experiments to assess the quality of PARAFD. First of all, we evaluated its time performance since efficiency is one of the main problems affecting the existing algorithms. The experiments described in Section 6.1 demonstrate that our implementation is usable in real scenarios. In Section 6.2 we tested the robustness of PARAFD by varying the dimensionality of the input data and the number of join connections. This evaluation confirms that our approach is scalable. Section 6.3 evaluates our technique for approximating full disjunctions; The experiments show a reduction of the time required for computing the approximation, and a limited loss of information. Finally, in Section 6.4, we evaluated the effectiveness of PARAFD by experimenting the full disjunction as a collection of documents for a text retrieval search engine.

**Implementation, environment.** We performed the experiments in a cluster of 6 virtual machines running Ubuntu 12.04. Each machine has 16 processors, 128 GB of RAM and 1 TB of storage. We implemented PARAFD using the Python interface of the Apache Spark framework.

**Dataset descriptions.** Three reference datasets [6] with complementary features have been used in our experiments. The Internet Movie Database (IMDB[12]) is a database of cinematographic data including more than 1.6 M tuples distributed in 6 relations and with a total size of 459 MBs. WIKIPEDIA dataset is a reduced version of the popular encyclopedia including six relations, more than 200k tuples and a total size of 391 MBs. MONDIAL[13] is a small dataset with a size of 16 MBs, 17K tuples (two orders of magnitude smaller than the IMDB dataset), but with a complex schema composed of 28 relations. By virtue of these characteristics, MONDIAL represents a meaningful test for the validation of full disjunction algorithms whose complexity mainly depends on the high number of data connections. IMDB can be thought as a typical business data source composed of a large amount of data and a simple schema. Finally WIKIPEDIA, containing the full text of articles, is a good option for the validation of keyword search systems.

### *6.1. Efficiency of the approach*

Three experiments have been performed to assess the efficiency of our approach. In a first experiment, we compared the execution time of our approach with respect to the existing algorithms and their parallelized versions described in Section 4. In a second experiment, we evaluated PARAFD time performance on the three reference datasets. Finally, we evaluated the hash star join implementation against the traditional left-deep join technique.

The comparison of the performance of the existing strategies for computing full disjunction has been performed by considering three subsets of the Mondial database. Table 1 shows the results of this experiment; Column 1 reports the subset of tables of Mondial used as configuration (input database), Column 2 reports the total number of tuples in the database and Column 3 reports the resulting number of tuples in the full disjunction of such database. For each data configuration, all proposed algorithms have been tested and their execution times are reported. Note that the results reported for $BiComNLOJ$

[12] https://www.imdb.com/

[13] https://www.dbis.informatik.uni-goettingen.de/Mondial/

| Configuration (Database $\mathcal{R}$) | Input dim. ($n^o$ of tuples of $\mathcal{R}$) | Output dim. ($n^o$ of tuples of $FD(\mathcal{R})$) | Algorithm version | Average time (h) and std |
|---|---|---|---|---|
| Mondial limited to IS_MEMBER, ORGANIZATION, COUNTRY | 8,399 | 8,021 | Original IncrementalFD | 11.42 (0.029) |
| | | | Table-driven naive parallelization | 6.50 (0.160) |
| | | | Two-phase parallelization | 2.23 (0.012) |
| | | | Block-based parallelization | 1.82 (0.037) |
| | | | Checkpoint-based version | 0.22 (0.005) |
| | | | BiComNLOJ | 1.65 (0.028) |
| | | | PARAFD | 0.024 (0.001) |
| Mondial limited to IS_MEMBER, ORGANIZATION, COUNTRY, CITY | 11,510 | 27,152 | Original IncrementalFD | 286.78 (4.709) |
| | | | Table-driven naive parallelization | 132.08 (2.256) |
| | | | Two-phase parallelization | 16.73 (0.084) |
| | | | Block-based parallelization | 8.89 (0.118) |
| | | | Checkpoint-based version | 3.24 (0.125) |
| | | | BiComNLOJ | 95.72 (1.584) |
| | | | PARAFD | 0.038 (0.001) |
| Mondial limited to IS_MEMBER, ORGANIZATION, COUNTRY, CITY, BORDERS | 11,830 | 96,964 | Original IncrementalFD | - |
| | | | Table-driven naive parallelization | - |
| | | | Two-phase parallelization | - |
| | | | Block-based parallelization | - |
| | | | Checkpoint-based version | 80.71 (0.017) |
| | | | BiComNLOJ | - |
| | | | PARAFD | 0.045 (0.001) |

Table 1: Efficiency with fragments of the MONDIAL Database. A mark "-" is reported for experiments not finished before the timeout (300 hours)

| Data source | Input dimension (tuples) | Output dimension (full disjunctions) | Average time (h) and std |
|---|---|---|---|
| IMDB | 1.6M | 4508339 | 0.193 (0.086) |
| Mondial | 17k | 282111161 | 2.264 (0.091) |
| Wikipedia | 200k | 375712 | 0.215 (0.015) |

Table 2: Efficiency of PARAFD in the reference datasets

refer to a parallelized version of the original algorithm we have implemented. To avoid noise and bias generated by other running applications and network failures, we have repeated the experiment 5 times for each algorithm. The table shows the average time and in brackets the standard deviation. A mark "-" shows experiments that did not finish before the timeout (300 hours).

The experiments show that PARAFD outperforms the existing techniques reducing the time required for computing full disjunctions until 4 orders of magnitude. Moreover, our implementation is the most scalable when the data increases. The time required has the same order of magnitude for all configurations. This behavior is not shared by the other techniques, where the times for completing the task largely varies with the input size. A further evaluation of PARAFD scalability is proposed in Section 6.2.

In the second experiment we measured the time to compute full disjunctions in the three reference datasets. As in the previous experiments, we repeated the computation 5 times to avoid bias. Table 2 shows the average time and the standard deviation measured. Despite its small size (17k tuples), MONDIAL generated the largest number of full disjunctions (more than $280 * 10^6$). This is due to the high number of foreign keys that generate more than 4k spanning trees. Our approach is the only technique among the ones tested able to compute this large number of full disjunctions (the other approaches fail after 300 hours of computation).

Finally, we evaluated the efficiency of the hash join algorithm implemented in PARAFD.

| | Left-deep outerjoins (s) | Extended hash join (s) | Time reduction (%) |
|---|---|---|---|
| Imdb | 496.50 | 269.84 | 45.65 |
| Wikipedia | 541.06 | 300.35 | 44.49 |
| Mondial | 157.59 | 121.00 | 23.22 |

Table 3: Comparison of extended hash join and left-deep outerjoins

The experiment consisted in comparing the execution time required by our hash star join algorithm to calculate the outerjoins between $n$ relations with respect to the one required by a sequence of left-deep outerjoins applied to respect the connected-prefix ordering. The approaches have been evaluated on a single tree selected from the schema graphs of the reference datasets. In particular, the schema graphs of the IMDB and WIKIPEDIA datasets generate only one tree. The spanning tree of maximum weight has been considered for the MONDIAL schema graph. The results of this evaluation are reported in Table 3. The hash star join technique performs better than a sequence of left-deep outerjoins: the percentage reduction of the execution times varies from about 25% to 40%. MONDIAL represents the dataset on which the performance of the proposed algorithm with respect to the considered baseline is smaller. This result is motivated by the fact that this dataset has a number of tuples and an overall size smaller compared to the other scenarios (see the dataset descriptions at the beginning of Section 6). This feature influences the times for data distribution among the cluster workers which impacts less on the entire process.

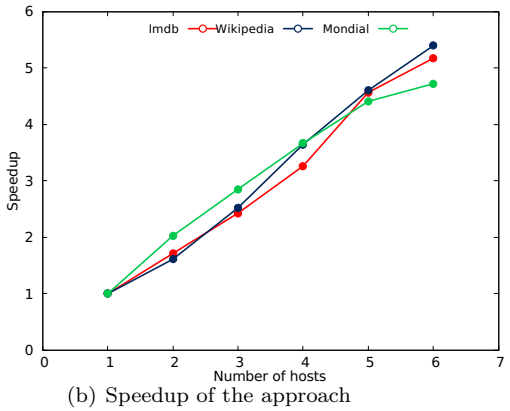
### *6.2. Robustness of the approach*

In this section we show the ability of our system to scale when the size of the data increases. In a first experiment, we executed PARAFD on the reference datasets by varying the number of active hosts within the considered cluster. The results are reported in Figure 10(a), where for each cluster configuration, we show the execution times. We observe that similar execution times are required in the IMDB and WIKIPEDIA databases (i.e., a minimum time of about 800 seconds with 6 active hosts and a maximum time exceeding 4000 seconds with a single host), while MONDIAL shows the highest execution times (i.e., about ten times higher). Then, in all scenarios, the execution times show a fairly linear trend with respect to the growth of the number of active hosts. A more intuitive representation of this trend is shown in Figure 10(b), where the speedup of our approach is reported for the three datasets. As can be seen, only the execution times obtained with 6 active hosts are slightly lower than an exact linear trend. This can be explained considering that the times for the distribution of data between the hosts increase.

In a second experiment we evaluated the execution time by varying the number of joining tuples in the input tables. For this purpose, synthetic datasets consisting of 3 tables, with a cardinality of 100k tuples and different distributions of the join connections were generated. In a first scenario we have evaluated the time performance with a balanced dataset, when the tables are connected only via one-to-one relations. In the other scenarios we built unbalanced distribution of joining tuples, where the cardinality of the tuples involved in one of the foreign/primary key relations is lower than the remaining two. In particular, each pair of tuples deriving from two over three tables was repeated with probability 0.9 a number of times equal to 25, 50, 100, 250 and 500 respectively.

| Num. hosts | IMDB time (s) | Wikipedia time (s) | Mondial time (s) |
|---|---|---|---|
| 1 | 4163.86 | 4475.78 | 38798.275 |
| 2 | 2430.33 | 2778.23 | 19140.744 |
| 3 | 1715.32 | 1775.54 | 13605.122 |
| 4 | 1276.07 | 1227.11 | 10583.29 |
| 5 | 911.55 | 971.89 | 8795.862 |
| 6 | 804.76 | 829.37 | 8215.42 |

(a) Comparison of execution times with different number of active hosts




(b) Speedup of the approach

Figure 10: Scalability of the approach

| | Balanced | Unbalanced | | | | |
|---|---|---|---|---|---|---|
| | | 25 | 50 | 100 | 250 | 500 |
| FD count | 100000 | 2041220 | 4060146 | 8125066 | 20322134 | 40624354 |
| Time (s) | 345.36 | 417.76 | 460.94 | 547.19 | 892.27 | 952.19 |

Table 4: Comparison of execution times with different number of join connections

Table 4 reports the number of full disjunction and the relative computation time for each of the synthetic datasets. The first scenario produces a number of full disjunction equal to the number of tuples of all three tables by construction. The time taken to calculate the full disjunction in this scenario is about 5 minutes. With the increase in the number of join associations between the tuples, the number of full disjunction also grows proportionally, but it is possible to notice that the execution time remains quite stable. It goes from 460.94 seconds to generate 40M full disjunction up to 952.19 seconds for 406M full disjunction. This experiment shows that the developed approach is able to scale even as the size of join connections between the tuples changes.

### *6.3. Effectiveness of the full disjunction approximation*

This section shows the experiments performed to evaluate the efficiency and the effectiveness of the technique for generating the approximated full disjunction. The efficiency has been evaluated by considering four degrees of approximation (i.e. the full disjunction is computed by considering 10%, 30%, 50%, 70% the overall number of spanning trees in the dataset) and measuring the number of relations in each approximation and the time required for their computation. Table 5 shows the results of our experiments executed in MONDIAL: as expected the number of full disjunction relations and the time required for their computation increases with the size of the approximation.

The ability of an approximating amount of relations in representing the entire full disjunction has been evaluated by observing if attribute values represented by the full disjunction relations are also existing in the approximated version. We tested the approach in three scenarios, where 100 attribute values, 100 pairs of values, and 100 triples have been randomly extracted. We computed, for each scenario, the precision, i.e., the

| Approximation | FDs | Time (h) |
|---|---|---|
| 10% | 70467668 | 0.787 |
| 30% | 126284216 | 1.264 |
| 50% | 187699268 | 1.492 |
| 70% | 217575090 | 1.639 |

Table 5: Efficiency of the approximated version of PARAFD

fraction of items that have been retrieved in the approximations, and the recall, i.e. the fraction of full disjunction containing the items in the approximation with respect to the full disjunction in the complete set. We performed the experiment by selecting a number of levels of approximation. Table 6 shows the results obtained. The value in brackets is the standard deviation. In all configurations high precision and recall levels are obtained.

| | 10% | | 30% | |
|---|---|---|---|---|
| | Prec. | Rec. | Prec. | Rec. |
| 1 term | 0.99 (0.10) | 0.30 (0.22) | 0.99 (0.10) | 0.48 (0.24) |
| 2 terms | 0.98 (0.14) | 0.28 (0.19) | 0.98 (0.14) | 0.48 (0.22) |
| 3 terms | 0.95 (0.22) | 0.25 (0.13) | 0.97 (0.17) | 0.45 (0.15) |
| | 50% | | 70% | |
| | Prec. | Rec. | Prec. | Rec. |
| 1 term | 0.99 (0.10) | 0.71 (0.22) | 0.99 (0.10) | 0.79 (0.19) |
| 2 terms | 0.98 (0.14) | 0.69 (0.21) | 0.98 (0.14) | 0.79 (0.16) |
| 3 terms | 0.98 (0.14) | 0.67 (0.14) | 0.98 (0.14) | 0.77 (0.11) |

Table 6: Effectiveness of the approximated version of PARAFD

*6.4. A keyword search system on relational databases based on a full disjunction*

In this section, we propose to consider the full disjunction as a collection of documents to be indexed by a text retrieval system (i.e., Lucene[14]). In this way, we implement a simple keyword search system on relational databases and we provide a measure of the effectiveness of the full disjunction in a real and challenging scenario. We experimented our idea against the benchmark proposed in [6], where IMDB, WIKIPEDIA and MONDIAL have been evaluated against 50 queries per source. The time required for indexing the data and the average time, the standard deviation, the minimum and the maximum time required to solve the queries is shown in Table 7. We observe that time required for solving the queries in most of the cases makes the simple approach implemented able to work real time with any optimization. Only in a few cases, the keyword queries are complex and require a large time to be completed.

Figure 11 shows the average precision of the first answer returned by PARAFD, compared with the other systems, in solving the keyword queries of the benchmark in the three datasets. Our approach largely outperforms the other systems in the scenarios related to the IMDB and WIKIPEDIA datasets, and works as the best other approaches in MONDIAL. The MONDIAL database schema is complex: the tables are connected via a large number of paths that form cycles. The result is that the same tuple is repeated more times in the full disjunction relations and the rank adopted by Lucene in some cases is not consistent with the one expected by the user.

[14] https://lucene.apache.org/

| Dataset | Indexing | Querying | | |
|---|---|---|---|---|
| | Time (s) | Average time (s) and std | Max time (s) | Min time (s) |
| MONDIAL | 22500 | 11.25 (30.17) | 192.12 | 0.05 |
| IMDB | 11520 | 20.47 (42.61) | 302.79 | 0.08 |
| WIKIPEDIA | 1250 | 160.23 (127.72) | 422.77 | 0.32 |

Table 7: Keyword search response times (in seconds)

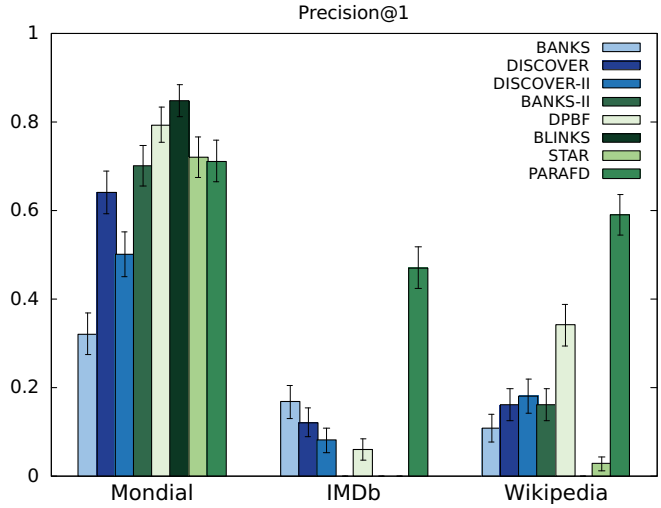



Figure 11: Average precision of the first answer of PARAFD compared with the benchmark [6]

## 7. Conclusion

In this paper we have presented PARAFD: a new approach, based on parallel computing techniques, for generating the full disjunction of a relational database. The same approach can also be used for obtaining an approximated full disjunction, where a limited number of full disjunction from the complete set is computed. For comparing our proposal with the state of the art, we have implemented $IncrementalFD$ and $BiComNLOJ$, two of the main algorithms available in the literature. The experiments demonstrate that PARAFD time performance outperforms existing approaches. The effectiveness of the approximated version has been also experimented, and we showed that it provides a good representation of the entire full disjunction. Finally, we have applied the full disjunction as a collection of documents to be indexed and retrieved by a search engine. The idea is to provide a basic answer to the problem of keyword search on relational database. Our idea has been compared against an existing benchmark, obtaining results with high recall and precision.

## References

[1] D. Maier, J. D. Ullman, M. Y. Vardi, On the foundations of the universal relation model, ACM Trans. Database Syst. 9 (2) (1984) 283–308.

[2] C. A. Galindo-Legaria, Outerjoins as disjunctions, in: Proceedings of the 1994 ACM SIGMOD international conference on Management of data, ACM Press, 1994, pp. 348–358.
[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012, 2012, pp. 15–28.
[4] S. Cohen, Y. Sagiv, An incremental algorithm for computing ranked full disjunctions, J. Comput. Syst. Sci. 73 (4) (2007) 648–668.
[5] S. Cohen, I. Fadida, Y. Kanza, B. Kimelfeld, Y. Sagiv, Full disjunctions: Polynomial-delay iterators in action, in: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006, 2006, pp. 739–750.
[6] J. Coffman, A. C. Weaver, An empirical performance evaluation of relational keyword search techniques, IEEE Trans. Knowl. Data Eng. 26 (1) (2014) 30–42.
[7] A. Rajaraman, J. D. Ullman, Integrating information by outerjoins and full disjunctions, in: Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada, 1996, pp. 238–248.
[8] V. Hristidis, L. Gravano, Y. Papakonstantinou, Efficient ir-style keyword search over relational databases, in: Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29, VLDB '03, VLDB Endowment, 2003, pp. 850–861.
[9] H. N. Gabow, E. W. Myers, Finding all spanning trees of directed and undirected graphs, SIAM J. Comput. 7 (3) (1978) 280–287.
[10] J. Aguilar-Saborit, V. Muntés-Mulero, C. Zuzarte, J. Larriba-Pey, Ad hoc star join query processing in cluster architectures, in: Data Warehousing and Knowledge Discovery, 7th International Conference, DaWaK 2005, Copenhagen, Denmark, August 22-26, 2005, Proceedings, 2005, pp. 200–209.
[11] I. Savnik, Index data structure for fast subset and superset queries, in: A. Cuzzocrea, C. Kittl, D. E. Simos, E. R. Weippl, L. Xu (Eds.), Availability, Reliability, and Security in Information Systems and HCI - IFIP WG 8.4, 8.9, TC 5 International Cross-Domain Conference, CD-ARES 2013, Regensburg, Germany, September 2-6, 2013. Proceedings, Vol. 8127 of Lecture Notes in Computer Science, Springer, 2013, pp. 134–148.
[12] R. De La Briandais, File searching using variable length keys, in: Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference, IRE-AIEE-ACM '59 (Western), ACM, New York, NY, USA, 1959, pp. 295–298.
[13] P. M. Amal, K. S. A. Kumar, An algorithm for $k^{th}$ minimum spanning tree, Electronic Notes in Discrete Mathematics 53 (2016) 343–354.
[14] X. Yang, C. M. Procopiuc, D. Srivastava, Summary graphs for relational database schemas, PVLDB 4 (11) (2011) 899–910.
[15] S. Bergamaschi, D. Ferrari, F. Guerra, G. Simonini, Y. Velegrakis, Providing insight into data source topics, J. Data Semantics 5 (4) (2016) 211–228.
[16] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004, pp. 137–150.
[17] G. P. Copeland, M. J. Franklin, G. Weikum, Uniform object management, in: Advances in Database Technology - EDBT'90. International Conference on Extending Database Technology, Venice, Italy, March 26-30, 1990, Proceedings, 1990, pp. 253–268.
[18] A. N. Wilschut, J. Flokstra, P. M. G. Apers, Parallelism in a main-memory DBMS: the performance of PRISMA/DB, in: 18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings., 1992, pp. 521–532.
[19] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, R. Rasmussen, The gamma database machine project, IEEE Trans. Knowl. Data Eng. 2 (1) (1990) 44–62.
[20] D. A. Schneider, D. J. DeWitt, Tradeoffs in processing complex join queries via hashing in multiprocessor database machines, in: 16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings., 1990, pp. 469–480.
[21] M. Chen, M. Lo, P. S. Yu, H. C. Young, Using segmented right-deep trees for the execution of pipelined hash joins, in: 18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings., 1992, pp. 15–26.
[22] M. Ziane, M. Zaït, P. Borla-Salamet, Parallel query processing with zigzag trees, VLDB J. 2 (3) (1993) 277–301.
[23] B. Liu, E. A. Rundensteiner, Revisiting pipelined parallelism in multi-join query processing, in:

Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005, 2005, pp. 829–840.

[24] K. Lee, Y. Lee, H. Choi, Y. D. Chung, B. Moon, Parallel data processing with mapreduce: a survey, SIGMOD Record 40 (4) (2011) 11–20.

[25] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, Y. Tian, A comparison of join algorithms for log processing in mapreduce, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, 2010, pp. 975–986.

[26] H. Yang, A. Dasdan, R. Hsiao, D. S. P. Jr., Map-reduce-merge: simplified relational data processing on large clusters, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007, 2007, pp. 1029–1040.

[27] D. Jiang, A. K. H. Tung, G. Chen, MAP-JOIN-REDUCE: toward scalable and efficient data analysis on large clusters, IEEE Trans. Knowl. Data Eng. 23 (9) (2011) 1299–1311.

[28] H. Han, H. Jung, H. Eom, H. Y. Yeom, Scatter-gather-merge: An efficient star-join query processing algorithm for data-parallel frameworks, Cluster Computing 14 (2) (2011) 183–197.

[29] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, S. Wu, Llama: leveraging columnar storage for scalable join processing in the mapreduce framework, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011, 2011, pp. 961–972.

[30] G. Zhou, Y. Zhu, G. Wang, Cache conscious star-join in mapreduce environments, in: 2nd International Workshop on Cloud Intelligence (colocated with VLDB 2013), Cloud-I '13, Riva del Garda, Trento, Italy, August 26, 2013, 2013, pp. 1:1–1:7.