Locality-aware Partitioning in Parallel Database Systems

Erfan Zamanian[†] Carsten Binnig[*†] Abdallah Salama[*]

[†] Brown University
Providence, USA

[*]Baden-Wuerttemberg Cooperative State University
Mannheim, Germany

ABSTRACT

Parallel database systems horizontally partition large amounts of structured data in order to provide parallel data processing capabilities for analytical workloads in shared-nothing clusters. One major challenge when horizontally partitioning large amounts of data is to reduce the network costs for a given workload and a database schema. A common technique to reduce the network costs in parallel database systems is to co-partition tables on their join key in order to avoid expensive remote join operations. However, existing partitioning schemes are limited in that respect since only subsets of tables in complex schemata sharing the same join key can be co-partitioned unless tables are fully replicated.

In this paper we present a novel partitioning scheme called predicate-based reference partition (or `PREF` for short) that allows to co-partition sets of tables based on given join predicates. Moreover, based on `PREF`, we present two automatic partitioning design algorithms to maximize data-locality. One algorithm only needs the schema and data whereas the other algorithm additionally takes the workload as input. In our experiments we show that our automated design algorithms can partition database schemata of different complexity and thus help to effectively reduce the runtime of queries under a given workload when compared to existing partitioning approaches.

1. INTRODUCTION

Motivation: Modern parallel database systems (such as SAP HANA [5], Greenplum [21] or Terradata [15]) and other parallel data processing platforms (such as Hadoop [22], Impala [1] or Shark [23]) horizontally partition large amounts of data in order to provide parallel data processing capabilities for analytical queries (e.g., OLAP workloads). One major challenge when horizontally partitioning data is to achieve a high data-locality when executing analytical queries since excessive data transfer can significantly slow down the query execution on commodity hardware [19].

SIGMOD'15, May 31 - June 04, 2015, Melbourne, VIC, Australia
 ACM 978-1-4503-2758-9/15/05 ...$15.00
http://dx.doi.org/10.1145/2723372.2723718.

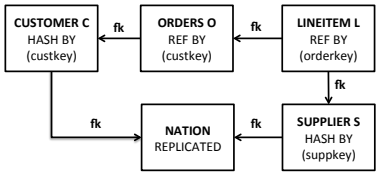


Figure 1: Partitioned TPC-H Schema (simplified)

A common technique to reduce the network costs in analytical workloads which was already introduced in the 1990's by the first parallel database systems is to co-partition tables on their join keys in order to avoid expensive remote join operations [8, 10]. However, in complex schemata with many tables this technique is limited to only subsets of tables, which share the same join key. Moreover, fully replicating tables is only desirable for small tables. Consequently, with existing partitioning schemes remote joins are typically unavoidable for complex analytical queries with join paths over multiple tables using different join keys.

Reference partitioning [9] (or `REF` partitioning for short) is a more recent partitioning scheme that co-partitions a table by another table that is referenced by an outgoing foreign key (i.e., referential constraint). For example, as shown in Figure 1, if table `CUSTOMER` is hash partitioned on its primary key `custkey`, then table `ORDERS` can be co-partitioned using the outgoing foreign key (fk) to the table `CUSTOMER` table. Thus, using `REF` partitioning, chains of tables linked via foreign keys can be co-partitioned. For example, table `LINEITEM` can also be `REF` partitioned by table `ORDERS`. However, other join predicates different from the foreign key or even incoming foreign keys are not supported by `REF` partitioning. For example, table `SUPPLIER` in Figure 1 can not be `REF` partitioned by the table `LINEITEM`.

Contributions: In this paper, we present a novel partitioning scheme called *predicate-based reference partitioning* (or `PREF` for short). `PREF` is designed for analytical workloads where data is loaded in bulks. The `PREF` partitioning scheme generalizes the `REF` partitioning scheme such that a table can be co-partitioned by a given join predicate that refers to another table (called partitioning predicate). In Figure 1, table `SUPPLIER` can thus be `PREF` partitioned by table `LINEITEM` using an equi-join predicate on the attribute `suppkey` as partitioning predicate. In order to achieve full data-locality with regard to the partitioning predicate, `PREF` might introduce duplicate tuples in different partitions. For example, when `PREF` partitioning the table `SUPPLIER` as described before and the same value for the `suppkey` attribute appears in multiple

partitions of the table LINEITEM, then the referencing tuple in table SUPPLIER will be duplicated to all corresponding partitions of SUPPLIER. That way, joins which use the partitioning predicate as join predicate can be executed locally per node. However, in the worst case, the PREF partitioning scheme might lead to full replication of a table. Our experiments show that this is only a rare case for complex schemata with a huge number of tables and can be avoided by our automatic partitioning design algorithms.

Furthermore, it is a hard problem to manually find the best partitioning scheme for a given database schema that maximizes data-locality using our PREF partitioning scheme. Existing automated design algorithms [14, 18, 20] are not aware of our PREF partitioning scheme. Thus, as a second contribution, we present two *partitioning design algorithms* that are aware of PREF. Our first algorithm is *schema-driven* and assumes that foreign keys in the schema represent potential join paths of a workload. Our second algorithm is *workload-driven* and additionally takes a set of queries into account. The main idea is to first find an optimal partitioning configuration separately for subsets of queries that share similar sets of tables and then incrementally merge those partitioning configuration. In our experiments, we show that by using the PREF partitioning scheme, our partitioning design algorithms outperform existing automated design algorithms, which rely on a tight integration with the database optimizer (i.e, to get the estimated costs for a given workload).

Outline: In Section 2, we present the details of the PREF partitioning scheme and discuss details about query processing and bulk loading. Afterwards, in Section 3, our schema-driven automatic partitioning design algorithm is presented. Section 4 then describes the workload-driven algorithm and discusses potential optimizations to reduce the search space. Our comprehensive experimental evaluation with the TPC-H [3] and the TPC-DS benchmark [2] is discussed in Section 5. We have chosen these two benchmarks as we wanted to show how our algorithms work for a simple schema with uniformly distributed data (TPC-H) and for a complex schema with skewed data (TPC-DS). Finally, we conclude with related work in Section 6 and a summary in Section 7.

2. PREDICATE-BASED REFERENCE PARTITIONING

In the following, we first present the details of our predicate-based reference partitioning scheme (or PREF for short) and then discuss important details of executing queries over PREF partitioned tables as well as bulk-loading those tables. In terms of notation we use capital letters for tables (e.g., table T) and small letters for individual tuples (e.g., tuple $t \in T$). Moreover, if a table T is partitioned into n partitions, the individual partitions are identified by $P_i(T)$ (with $1 \leq i \leq n$).

2.1 Definition and Terminology

The PREF partitioning scheme is defined as follows:

DEFINITION 1 (PREF PARTITIONING SCHEME). *If a table S is partitioned into n partitions using an arbitrary horizontal partitioning-scheme, then table R is* PREF *partitioned by that table S and a given partitioning predicate p, iff (1) for all $1 \leq i \leq n$, $P_i(R) = \{r | r \in R \wedge (\exists s \in P_i(S) | p(r,s))\}$ holds and (2) $\forall r \in R | (\exists r \in P_i(R), 1 \leq i \leq n)$. In* PREF *we call S the referenced table and R the referencing table. The referenced table could be again* PREF *partitioned. The seed table of a* PREF *partitioned table R is the first table T in the path of the partitioning predicates that is not* PREF *partitioned.*

Condition (1) in the definition above means that a tuple $r \in R$ is in a partition $P_i(R)$ if there exists at least one tuple $s \in P_i(S)$ that satisfies the given partition predicate p (i.e., $p(r,s)$ evaluates to true) for the given i. A tuple s that satisfies $p(r,s)$ is called partitioning partner. Therefore, if p is satisfied for tuples in different partitions of S, then a copy of r will be inserted to all these partitions which leads to duplicates (i.e., redundancy). Moreover, condition (2) means that each tuple $r \in R$ must be assigned to at least one partition (even if there exists no tuple $s \in P_i(S)$ in any partition of S that satisfies $p(r,s)$). In order to satisfy condition (2), we assign all tuples $r \in R$ that do not have a partitioning partner in S in a round-robin fashion to the different partitions $P_i(R)$ of R.

As mentioned before, any partitioning scheme can be used (e.g., hash, range, round-robin or even PREF) for the *referenced table*. For simplicity but without loss of generality, we use only the HASH and PREF partitioning scheme in the remainder of the paper. Moreover, only simple equi-join predicates (as well as conjunctions of simple equi-join predicates) are supported as partitioning predicates p since other join predicates typically result in full redundancy of the PREF partitioned table (i.e., a tuple is then likely to be assigned to each partition of R).

Example: Figure 2 shows an example of a database before partitioning (upper part) and after partitioning (lower part). In the example, the table LINEITEM is hash partitioned and thus has no duplicates after partitioning. The table ORDERS (o) is PREF partitioned by table LINEITEM (l) using a partitioning predicate on the join key (orderkey); i.e., ORDERS is the referencing table and LINEITEM the referenced table as well as the seed table. For the table ORDERS, the PREF partitioning scheme introduces duplicates to achieve full data-locality for a potential equi-join over the join key (orderkey). Furthermore, the table CUSTOMER (c) is PREF partitioned by ORDERS using the partitioning predicate on the join key (custkey); i.e., CUSTOMER is the referencing table and ORDERS the referenced table whereas LINEITEM is the seed table of the CUSTOMER table. Again, PREF partitioning the CUSTOMER table results in duplicates. Moreover, we can see that the customer (custkey=3), who has no order, is also added to the partitioned table CUSTOMER (in the first partition).

Thus, by using the PREF partitioning scheme, all tables in a given join path of a query can be co-partitioned as long as there is no cycle in the query graph. Finding the best partitioning scheme for all tables in a given schema and workload that maximizes data-locality under the PREF scheme, however, is a complex task and will be discussed in Sections 3 and 4.

For query processing, we create two additional bitmap indexes when PREF partitioning a table R by S: The first bitmap index dup indicates for each tuple $r \in R$ if it is the first occurrence (indicated by a 0 in the bitmap index) or if r is a duplicate (indicated by a 1 in the bitmap index). That way duplicates that result from PREF partitioning can be easily eliminated during query processing, as will be explained in

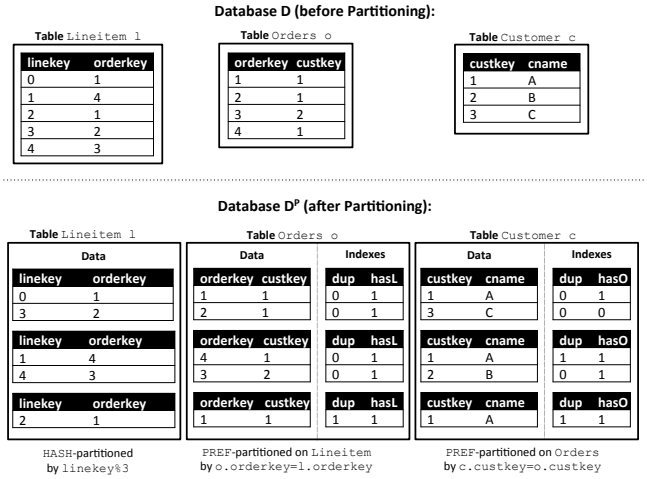


Figure 2: A PREF partitioned Database

more details in Section 2.2. The second index `hasS` indicates for each tuple $r \in R$ if there exists a tuple $s \in S$ which satisfies $p(r, s)$. That way, anti-joins and semi-joins can be optimized. The example in Figure 2 shows these indexes for the two `PREF` partitioned tables. Details about how these indices are used for query processing is discussed in the following Section 2.2.

2.2 Query Processing

In the following, we discuss how queries need to be rewritten for correctness, if `PREF` partitioned tables are included in a given SQL query Q. This includes adding operations for eliminating duplicates resulting from `PREF` partitioned tables and adding re-partitioning operations for correct parallel query execution. Furthermore, we also discuss rewrites for optimizing SQL queries (e.g., to optimize queries with anti-joins or outer joins). All these rewrite rules are applied to a compile plan P of query Q. Currently, our rewrite rules only support SPJA queries (Selection, Projection, Join and Aggregation), while nested SPJA queries are supported by rewriting each SPJA query block individually.

Rewrite Process: The rewrite process is a bottom-up process, which decides for each operator $o \in P$ if a distinct operation or a re-partitioning operation (i.e., a shuffle operation) must be applied to its input(s) before executing the operator o. Note that our distinct operator is not the same as the SQL DISTINCT operator. Our distinct operator eliminates only those duplicates which are generated by our `PREF` scheme. Duplicates from `PREF` partitioning can be eliminated using a disjunctive filter predicate that uses the condition `dup=0` for each `dup` attribute of a tuple in an (intermediate) result. A normal SQL DISTINCT operator, however, can still be executed using the attributes of a tuple to find duplicates with the same values. In the rest of the paper, we always refer to the semantics of our distinct operator. Moreover, the re-partitioning operator also eliminates duplicates resulting from `PREF` partitioning before shuffling tuples over the network.

In order to decide if a distinct operation or a re-partitioning operation must be applied, the rewrite process annotates two properties to each intermediate result of an operator $o \in P$. In the following, we also use the variable o to refer to the intermediate result produced by operator o.

- $Dup(o)$: defines if the (intermediate) result o is free of duplicates resulting from `PREF` partitioning ($Dup(o) = 0$) or not ($Dup(o) = 1$). For tables, we use the same notation; i.e., $Dup(T)$ defines whether table T contains duplicates due to `PREF` partitioning or not. For a hash partitioned table T, we get $Dup(T) = 0$.
- $Part(o)$: defines a partitioning scheme for the (intermediate) result o including the partitioning method $Part(o).m$ (`HASH`, `PREF`, or `NONE` if neither of the other schemes holds), the list of partitioning attributes $Part(o).A$ and the number of partitions $Part(o).c$. Again, for tables we use the same notation; i.e., $Part(T)$ defines the partitioning scheme of table T.

In the following, we discuss the rewrite rules for each type of operator of an SPJA query individually. Moreover, we assume that the last operator of a plan P is always projection operator that can be used to eliminate duplicates resulting from `PREF` partitioning. We do not discuss the selection operator since neither additional duplicate eliminations nor re-partitioning operators need to be added to its input (i.e., the selection operator can be executed without applying any of these rewrites).

Inner equi-join $o = (o_{in1} \bowtie_{o_{in1}.a_1 = o_{in2}.a_2} o_{in2})$: The only re-write rule, we apply to an inner equi-join, is to add additional re-partitioning operators over its inputs o_{in1} and o_{in2}. In the following, we discuss three cases when no re-partitioning operator needs to be added.

(1) The first case holds, if both inputs are hash partitioned and they use the same number of partitions (i.e., $Part(o_{in1}).c = Part(o_{in2}).c$ holds). Moreover, $Part(o_{in1}).A = [a_1]$ and $Part(o_{in2}).A = [a_2]$ must hold as well (i.e., the join keys are used as partitioning attributes).

(2) The second case holds, if $Part(o_{in1}).m =$ `HASH` and $Part(o_{in2}).m =$ `PREF` whereas the join predicate $a_1 = a_2$ must be the partitioning predicate of the `PREF` scheme. Moreover, the partitioning scheme $Part(o_{in1})$ must be the one used for the seed table of the `PREF` scheme $Part(o_{in2})$.

(3) The third case holds, if $Part(o_{in1}).m =$ `PREF` and $Part(o_{in2}).m =$ `PREF` whereas the join predicate $a_1 = a_2$ must be the partitioning predicate of the `PREF` schema of one input (called referencing input). The other input is called referenced input. Moreover, both `PREF` schemes must reference the same seed table.

For example, case (2) above holds for a join operation ($l \bowtie_{l.linekey=o.linekey} o$) over the partitioned database in Figure 2. Moreover, case (3) holds for a join operation ($o \bowtie_{o.custkey=c.custkey} c$) over the same schema.

Otherwise, if none of these three cases holds, the rewrite procedure applies re-partitioning operators to make sure that both inputs use a hash partitioning scheme where the join key is the partitioning attribute and both schemes use the same number of partitions. The re-partitioning operator also eliminates duplicates resulting form a `PREF` scheme as discussed before. If one input is already hash partitioned (using the join key as partitioning attribute), then we only need to re-partition the other input accordingly.

After discussing the rewrite rules, we now present how the properties $Dup(o)$ and $Part(o)$ are set by the rewrite procedure. If we add a re-partitioning operation as discussed before, then we use the hash partitioning scheme

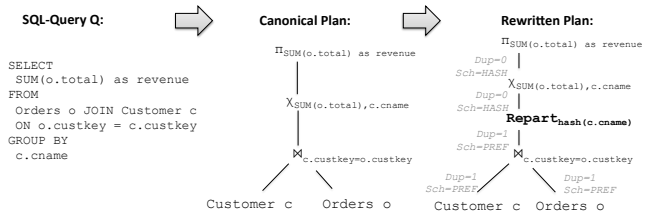


Figure 3: Rewrite Process for Plan P

of the re-partitioning operator to initialize $Part(o)$ and set $Dup(o) = 0$ since we eliminate duplicates. In case that we do not add a re-partitioning operation (i.e., in the cases 1-3 before), we initialize $Part(o)$ as follows: In case (1), we set $Part(o)$ to be the hash partitioning scheme of one of the inputs (remember, that both inputs use the same partitioning scheme) and $Dup(o) = 0$ since hash partitioned tables never contain duplicates. In cases (2) and (3), we use the `PREF` scheme of the referenced input to initialize $Part(o)$. Moreover, in case (2), we always set $Dup(o) = 0$. In case (3), we set $Dup(o) = 0$ if the referenced input has no duplicates. Otherwise, we set $Dup(o) = 1$.

For example, for the intermediate result of the join $(c \bowtie_{c.custkey=o.custkey} o)$ where case (3) holds, the rewrite procedure initializes $Part$ to be the `PREF` scheme of the `ORDERS` table and sets Dup to 1.

Other joins: While equi-joins can be executed on partitioned inputs (as discussed before), other joins such as cross products $o = (o_{in1} \times o_{in2})$ and theta joins $o = (o_{in1} \bowtie_p o_{in2})$ with arbitrary join predicates p need to be executed as remote joins that ships the entire smaller relation to all cluster nodes. For these joins, we set $Part(o).m =$ `NONE` . Moreover, we also eliminate duplicates in both inputs and set $Dup(o) = 0$.

Furthermore, an outer join can be computed as the union of an inner equi-join and an anti-join. An efficient implementation for anti-joins is presented at the end of this section.

Aggregation $o = \chi_{GrpAtts,AggFuncs}(o_{in})$: If the partition scheme of the input operator o_{in} is hash partitioned and if the condition $GrpAtts.startWith(Part(o_{in}).A)$ (i.e., the list of group-by attributes starts with or is the same as the list of partitioning attributes), then we do not need to re-partition the input. Otherwise, a re-partitioning operator is added that hash partitions the input o_{in} by the $GrpAtts$. Moreover, if $Dup(o_{in}) = 1$ holds, the re-partitioning operator also eliminates duplicates (as described before). Finally, the rewrite process sets $Part(o)$ to $Part(o_{in})$ if no re-partitioning operator is added. Otherwise, it sets $Part(o)$ to the hash partitioning scheme used for re-partitioning. Moreover, in any case it sets $Dup(o) = 0$.

Figure 3 shows an example of an aggregation query over the partitioned database shown in Figure 2. In that example, the output of the join is `PREF` partitioned and contains duplicates (as already discussed before). Thus, the input of the aggregation must be re-partitioned using a hash partitioning scheme on the group-by attribute *c.cname* (which is used as the partitioning scheme of its output). Moreover, the re-partitioning operators eliminates the duplicates resulting from `PREF`.

Projection $o = \pi_{Atts}(o_{in})$: For the projection operator the input o_{in} is never re-partitioned. However, if $Dup(o_{in}) = 1$ we add a distinct operation on input o_{in} that eliminates duplicates using the `dup` indexes. For this operator, we set $Part(o) = o_{in}$ and $Dup(o) = 0$.

Further Rewrites for Query Optimization: Further rewrite rules can be applied for query optimization when joining a `PREF` partitioned table R with a referenced table S on p using the index `hasS`: (1) An anti-join over R and S can be rewritten by using a selection operation with the filter predicate $hasS = 0$ on R without actually joining S. (2) A semi-join over R and S can be rewritten by using a selection operation with the filter predicate $hasS = 1$ on R without actually joining S.

2.3 Bulk Loading

As discussed before, the `PREF` scheme is designed for data warehousing scenarios where new data is loaded in bulks. In the following, we discuss how inserts can be executed over a `PREF` partitioned table R that references a table S. We assume that the referenced table S has already been bulk loaded.

In order to insert a new tuple r into table R, we have to identify those partitions $P_i(R)$ into which a copy of a tuple r must be inserted. Therefore, we need to identify those partitions $P_i(S)$ of the referenced table S that contain a partitioning partner (i.e., a tuple s which satisfies the partitioning predicate p for the given tuple r). For example, in Figure 2 table `CUSTOMER` is `PREF` partitioned referencing table `ORDERS`. When inserting a customer tuple with $custkey = 1$ into table `CUSTOMER`, a copy must be inserted into all three partitions since all partitions of `ORDERS` have a tuple with $custkey = 1$.

For efficiently implementing the insert operation of new tuples without executing a join of R with S, we create a partition index on the referenced attribute of table S. The partition index is a hash-based index that maps unique attribute values to partition numbers i. For example, for the table `ORDERS` schema in Figure 2, we create a partition index on the attribute $custkey$ that maps e.g. $custkey = 1$ to partitions 1 to 3. We show in our experiments in Section 5 that partition indexes help to efficiently execute bulk loading of new tuples.

Finally, updates and deletes over a `PREF` partitioned table are applied to all partitions. However, we do not allow that updates modify those attributes used in a partitioning predicate of a `PREF` scheme (neither in the referenced nor in the referencing table). Since join keys are typically not updated in data warehousing scenarios this restriction does not limit the applicability of `PREF`.

3. SCHEMA-DRIVEN AUTOMATED PARTITIONING DESIGN

In this section, we present our schema-driven algorithm for automated partitioning design. We first discuss the problem statement and then give a brief overview of our solution. Afterwards, we present important details on how we maximize data-locality while minimizing data-redundancy.

3.1 Problem Statement and Overview

The *Problem Statement* can be formulated as the following optimization problem: Given a schema S (including referential constraints) and the non-partitioned database D, define

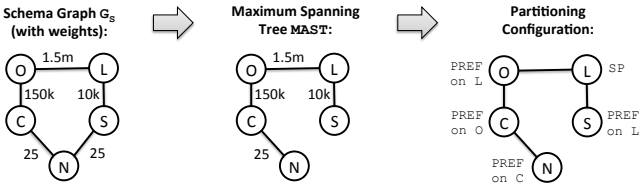


Figure 4: Schema-driven Partitioning Design

a partitioning scheme (`HASH` or `PREF`) for each table $T \in S$ (called partitioning configuration) such that data-locality in the resulting partitioned database D^P is maximized with regard to equi-join operations over the referential constraints, while data-redundancy is minimized. In other words, while the main optimization goal is maximizing data-locality under the given partitioning schemes, among those materialization configurations with the same highest data-locality, the one with the minimum data-redundancy should be chosen.

Note that in the above problem statement, we do not consider full replication as a possible choice for a table. The reason is that full replication is only desirable for small tables, while `PREF` can find a middle ground between partitioning and full replication for other tables that can not be fully replicated. Furthermore, small tables that are candidates for full replication can be excluded from the given database schema before applying our design algorithm. In order to solve the before mentioned optimization problem, our algorithm executes the following three steps.

The *first step* is to create an undirected labeled and weighted graph $G_S = (N, E, l(e \in E), w(e \in E))$ for the given schema S (called *schema graph*). While a node $n \in N$ represents a table, an edge $e \in E$ represents a referential constraint in S. Moreover, the labeling function $l(e \in E)$ defines the equi-join predicate for each edge (which is derived from the referential constraint) and the weighting function $w(e \in E)$ defines the network costs if a remote join needs to be executed over that edge. The weight $w(e \in E)$ of an edge is defined to be the size of the smaller table connected to the edge e. The intuition behind this is that the network costs of a potential remote join over an edge e depend on the size of the smaller table, since this table is typically shipped over the network. It is clear that we ignore the selectivity of more complex queries (with selection operators and multiple joins) and thus $w(e \in E)$ only represents an upper bound. However, our experiments show that $w(e \in E)$ is a good proxy to represent the total workloads costs even for workloads with complex queries. Figure 4 (left hand side) shows the schema graph resulting from our simplified version of the TPC-H schema for scaling factor $SF = 1$.

As a *second step*, we extract a subset of edges E_{co} from G_S that can be used to co-partition all tables in G_S such that data-locality is maximized. For a given connected G_S, the desired set of edges E_{co} is the maximum spanning tree (or `MAST` for short). The reason of using the `MAST` is that by discarding edges with minimal weights from the G_S, the network costs of potential remote joins (i.e., over edges not in the `MAST`) are minimized and thus data-locality as defined above is maximized. A potential result of this step is shown in Figure 4 (center).

Typically, there exists more than one MAST with the same total weight for a connected G_S. For example, in Figure 4, instead of discarding the edge between `SUPPLIER` and `NATION`, one could also discard the edge between `CUSTOMER` and `NATION` since this edge has the same weight. If different `MAST`s with the same total weight exist, then the following step must be applied for each `MAST` individually.

Finally, in the *last step* we enumerate all possible partitioning configurations that can be applied for the `MAST` to find out which partitioning configuration introduces the minimum data-redundancy. Minimizing data-redundancy is important since this has direct effect on the runtime of queries (even if we can achieve maximal data-locality). The partitioning configurations, which we enumerate in our algorithm, all follow the same pattern: one table in the `MAST` is selected to be the seed table which uses a hash partitioning scheme. In general, it could use any of the existing partitioning schemes such as hash, round-robin, or range partitioning. As partitioning attribute, we use the join attribute in the label $l(e)$ of the edge $e \in E$, which is connected to the node representing the seed table and has the highest weight $w(e)$. All other tables are recursively `PREF` partitioned on the seed table using the labels of the edges in the `MAST` as partitioning predicates. Figure 4 (right hand side) shows one potential partitioning configuration for the `MAST`, which uses the `LINEITEM` table as the seed table.

3.2 Maximizing Data-Locality

Data-locality (DL) for a given schema graph G_S and the subset of edges E_{co} used for co-partitioning is defined as follows:

$$DL = \frac{\sum_{e \in E_{co}} w(e)}{\sum_{e \in E} w(e)}$$

While $DL = 1$ means that E_{co} contains all edges in G_S (i.e., no remote join is needed), $DL = 0$ means that E_{co} is empty (i.e., no table is co-partitioned by any other table). For example, if we hash partition all tables of a schema on their primary keys, then data-locality will be 0 (as long as the tables do not share the same primary key attributes).

In order to maximize data-locality for a given schema graph G_S that has only one connected component, we extract the maximum spanning tree `MAST` based on the given weights $w(e \in E)$. The set of edges in the `MAST` represents the desired set E_{co} since adding one more edge to a `MAST` will result in a cycle which means that not all edges can be used for co-partitioning. If G_S has multiple connected components, we extract the `MAST` for each connected component. In this case E_{co} represents the union over the edges of all maximum spanning trees.

One other solution (instead of extracting the `MAST`) is to duplicate tables (i.e., nodes) in the G_S in order to remove cycles and allow one table to use different partitioning schemes. However, join queries could still potentially require remote joins. For example, if we duplicate table `NATION` in the G_S of Figure 4 (left hand side), we can co-partition one copy of `NATION` by `CUSTOMER` and one copy of `NATION` by `SUPPLIER`. However, a query using the join path $\texttt{C} - \texttt{N} - \texttt{S}$ then still needs a remote join either from over the edge $C - N$ or the edge $N - S$. Therefore, in our schema-driven algorithm we do not duplicate nodes at all.

3.3 Minimizing Data-Redundancy

The next step after maximizing data-locality is to find

Listing 1: Enumerating Partitioning Configurations

```
1  function findOptimalPC(MAST mast, Database D){
2    PartitionConfig optimalPC;
3    optimalPC.estimatedSize = MAX_INT;
4
5    for(each node n_ST in N(mast)){
6      //build new PC based on seed table
7      PartitionConfig newPC;
8      newPC.addScheme(n_ST, SP);
9      addPREF(mast, n_ST, newPC);
10
11     //estimate size of newPC
12     estimateSize(newPC, mast, D);
13     if(newPC.estimatedSize<optimalPC.estimatedSize)
14       optimalPC = newPC;
15   }
16   return optimalPC;
17 }
18
19 //recursively PREF partition tables
20 function addPREF(MAST mast, Node referring,
21   PartitionConfig pc){
22   for(each node ref connected
23     to referring by edge e in mast){
24     if(pc.containsScheme(ref))
25       continue;
26     newPC.addScheme(ref, PREF on referring by l(e));
27     addPREF(mast, ref, pc);
28   }
29 }
```

a partitioning configuration for all tables in the schema S, which minimizes data-redundancy in the partitioned database D^P. Therefore, we first define data-redundancy (DR) as follows:

$$DR = \frac{|D^P|}{|D|} - 1 = \frac{\sum_{T \in S} |T^P|}{\sum_{T \in S} |T|} - 1$$

While $|D^P|$ represents the size of the database after partitioning, $|D|$ represents the original size of the database before partitioning. $|D^P|$ is defined to be the sum of sizes of all tables $T \in S$ after partitioning (denoted by T^P). Consequently, $DR = 0$ means that no data-redundancy was added to any table after partitioning, while $DR = 1$ means that 100% data-redundancy was added after partitioning (i.e., each tuple in D exists in average twice in D^P). Fully replicating each table to all n nodes of a cluster thus results in data-redundancy $n - 1$.

In Listing 1, we show the basic version of our algorithm to enumerate different partitioning configurations (PCs) for a given `MAST`. For simplicity (but without loss of generality), we assume that the schema graph G_S has only one connected component with only one `MAST`. Otherwise, we can apply the enumeration algorithm for each `MAST` individually.

The enumeration algorithm (function `findOptimalPC` in Listing 1) gets a `MAST` and a non-partitioned database D as input and returns the optimal partitioning configuration for all tables in D. The algorithm therefore analyzes as many partitioning configurations as we have nodes in the `MAST` (line 5-15). Therefore, we construct partitioning configurations (line 7-9) that follow the same pattern: one table is used as the seed table that is partitioned by one of the seed partitioning schemes (or `SP` for short) such as hash partitioning and all other tables are recursively `PREF` partitioned on the edges of the `MAST` (see function `addPREF`). For each partitioning configuration $newPC$, we finally estimate the size of the partitioned database when applying $newPC$ and compare it to the optimal partitioning configuration so far (line 12-14). While seed tables in our partitioning design algorithms never contain duplicate tuples, `PREF` partitioned tables do. In order to estimate the size of a database after partitioning, the expected redundancy in all tables which are `PREF` partitioned must be estimated. Redundancy is cumulative, meaning that if a referenced table in the `PREF` scheme contains duplicates, the referencing table will inherit those duplicates as well. For example, in Figure 2 the duplicate orders tuple with $orderkey = 1$ in the `ORDERS` table results in a duplicate customer tuple with $custrkey = 1$ in the referencing `CUSTOMER` table. Therefore, in order to estimate the size of a given table, all referenced tables up to the seed (redundancy-free) table must be considered. The details of the size estimation of tables after partitioning are explained in Appendix A.

3.4 Redundancy-free Tables

As described in Section 3.3, the final size of a given table after partitioning is determined by the redundancy factors of all the edges from the seed table to the analyzed table. In complex schemata with many tables, this might result in full or near full redundancy for `PREF` partitioned tables. This is because only one table is determined by the algorithm to be the seed table, while all other tables are `PREF` partitioned. In order to remedy this problem, our enumeration algorithm can additionally take user-given constraints as input which disallows data-redundancy for individual tables. Therefore, we adopt the enumeration algorithm described in Section 3.3 as follows: (1) We also enumerate partitioning configurations which can use more than one seed table. We start with configurations with one seed table and increase the number up to $|S|$ seed tables until we satisfy the user-given constraints. Since the maximal data-locality for a `MAST` monotonically decreases with an increasing number of seed tables, we can stop the enumeration early once we find a partitioning configuration that satisfies the user-given constraints. This scheme, will be the partitioning scheme with the maximal data-locality that also satisfies the user-given constraints. (2) We prune partitioning configurations early that add data-redundancy for tables where a user-given constraint disallows data-redundancy. That means for tables where we disallow data-redundancy, we can either use a seed partitioning scheme or a `PREF` partitioning scheme whose partition predicate refers to the primary key of a table that has no data-redundancy.

4. WORKLOAD-DRIVEN AUTOMATED PARTITIONING DESIGN

In this section, we discuss our workload-driven automated partitioning design algorithm. Again, we start with the problem statement and then give a brief overview of our solution. Afterwards, we discuss the details of our algorithm.

4.1 Problem Statement and Overview

The *Problem Statement* can be formulated as the following optimization problem: Given a schema S, a workload $W = \{Q_1, Q_2, ..., Q_n\}$ and the non-partitioned database D, define a partitioning scheme (`HASH` or `PREF`) for the tables

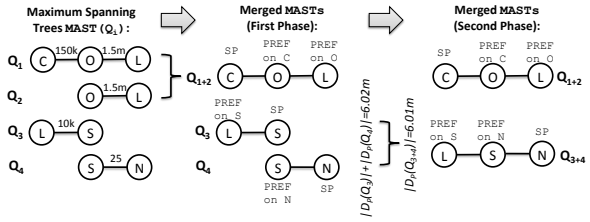


Figure 5: Workload-driven Partitioning Design

used by each query $Q_i \in W$ (called partitioning configuration) such that data-locality is maximized for each query Q_i individually, while data-redundancy is globally minimized for all $Q_i \in W$. Like schema-driven partitioning, the main optimization goal is to maximize data-locality under the given partitioning schemes; data-redundancy is only subordinate. In order to solve this optimization problem our algorithm executes the following three steps.

In the *first step* our algorithm creates a separate schema graph $G_S(Q_i)$ for each query $Q_i \in W$ where edges represent the join predicates in a query. Afterwards, we compute the $\texttt{MAST}(Q_i)$ for each $G_S(Q_i)$. That way, data-locality for each query $Q_i \in W$ is maximized, since one optimally partitioned minimal database $D^P(Q_i)$ could be generated for each query individually. However, this would result in a very high data-redundancy since individual tables will most probably exist several times (using different partitioning schemes for different queries). For example, Figure 5 (left hand side) shows the MASTs resulting from four different queries in a workload $W = \{Q_1, Q_2, Q_3, Q_4\}$. Again, if different MASTs with the same total weight exist for one query, we can keep them all to find the optimal solution in the following steps.

In the *second step*, we merge MASTs of individual queries in order to reduce the search space of the algorithm. Given the MASTs, the merge function creates the union of nodes and edges in the individual MASTs.In this phase we merge a $\texttt{MAST}(Q_j)$ into a $\texttt{MAST}(Q_i)$ if the MAST of Q_j is fully contained in the MAST of Q_i (i.e. $\texttt{MAST}(Q_i)$ contains all nodes and edges with the same labels and weights of $\texttt{MAST}(Q_j)$). Thus, no cycles can occur in this merge phase. The merged MAST is denoted by $\texttt{MAST}(Q_{i+j})$ If $\texttt{MAST}(Q_j)$ is fully contained in different MASTs, we merge it into one of these MASTs. Moreover, at the end of the first merging phase, we determine the optimal partitioning configuration and estimate the total size of the partitioned database for each merged MAST (using function `findOptimalPC` in Listing 1). Figure 5 (center) shows a potential result of the first merging phase. This step effectively reduces the search space for the subsequent merging phase.

In the *last step* (i.e. a second merge phase), we use a cost-based approach to further merge MASTs. In this step, we only merge $\texttt{MAST}(Q_j)$ into $\texttt{MAST}(Q_i)$ if the result is acyclic and if we do not sacrifice data-locality while reducing data-redundancy (i.e., if $|D^P(Q_{i+j})| < |D^P(Q_i)|+|D^P(Q_j)|$ holds). Figure 5 (right hand side) shows a potential result of the second merging phase. In this example MAST of Q_3 and Q_4 are merged since the size of the resulting database $D^P(Q_{3+4})$ after merging is smaller than the sum of sizes of the individual partitioned databases $D^P(Q_3)+D^P(Q_4)$. For query execution, a query can be routed to the MAST which contains the query and which has minimal data-redundancy for all tables read by that query.

4.2 Maximizing Data-Locality

In order to maximize data-locality for a given workload W, we first create a separate schema graph $G_S(Q_i)$ for each query $Q_i \in W$ as described before. The schema graph for the workload-driven algorithm is defined the same way as described in Section 3 as $G_S = (N, E, l(e \in E), w(e \in E))$ and can be derived from the query graph of a query Q_i: A query graph is defined in the literature as an undirected labeled graph $G_Q = (N, E, l(e \in E))$ where each node $n \in N$ represents a table (used by the query). An edge $e \in E$ represents a join predicate between two tables while the labeling function $l(e \in E)$ that returns the join predicate for each edge.

Currently, when transforming a query graph $G_Q(Q_i)$ into an equivalent schema graph $G_S(Q_i)$, we only consider those edges which use an equi-join predicate as label. Note that this does not mean that queries in workload W can only have equi-join predicates. It only means that edges with non-equi join predicates are not added to the schema graph since these predicates result in high data-redundancy anyway when used for co-partitioning tables by PREF as discussed in Section 2. Moreover, for creating the schema graph G_S, a weighting function $w(e \in E)$ needs to be defined for the G_S. This is trivial since the table sizes are given by the non-partitioned database D that is an input to the workload-driven algorithm as well. Note that in addition to table sizes, edge weights G_S could also reflect costs of a query optimizer to execute the join, if these information items are provided. However, then the merging function would need to be more complex (i.e., a simple union of nodes and edges is not enough since the same edge could have different weights). In the following, we assume that edge weights represent table sizes.

Once the schema graph $G_S(Q_i)$ is created for each query $Q_i \in W$, we can derive the maximum spanning tree $\texttt{MAST}(Q_i)$ for each $G_S(Q_i)$. The $\texttt{MAST}(Q_i)$ represents the set of edges $E_{co}(Q_i)$ that can be used for co-partitioning tables in Q_i. All edges that are in the query graph of Q_i but not in the $\texttt{MAST}(Q_i)$ will result in remote joins. Data-locality DL for a query is thus defined in the same way as before in Section 3.2 as the fraction of the sum of weights in $E_{co}(Q_i)$ and the sum of weights for all edges in $G_S(Q_i)$. As shown in Section 3 using the edges of a MAST for co-partitioning maximizes data-locality unless we additionally allow to duplicate tables (i.e., nodes) in order to remove cycles in $G_S(Q_i)$. Moreover, in contrast to the schema-driven algorithm, if a connected G_S has different MASTs with the same total weight, our algorithm additionally finds the optimal partitioning configuration for each of the MASTs and estimates the size of the partitioned database as shown in Listing 1. For the subsequent merging phase, we only keep that MAST which results in a partitioned database with minimal estimated size.

4.3 Minimizing Data-Redundancy

Merging the MASTs of different queries is implemented in two steps as described before: using heuristics in the first merge phase to effectively reduce the search space for the second cost-based merge phase to further reduce data-redundancy. The result after both merging phases is a set of MASTs and an optimal partitioning configuration for each MAST. If a table appears in different MASTs using different partitioning schemes in the partitioning configuration, we duplicate the table in the final partitioned database D^P that

we create for all MASTs. However, if a table appears in different MASTs and uses the same partitioning scheme we do not duplicate this table in D^P. Data-redundancy for a set of MASTs is thus defined as a fraction of the sum of all partitioned tables and the size of the non-partitioned D. In the following, we only discuss the cost-based merging in detail since the first merge step is trivial.

For cost-based merging, we first define the term *merge configuration*. A merge configuration is a set of merge expressions, which defines for each query Q_i in a given set of queries if the MASTs are merged or not: Q_{i+j} is a merge expression which states that the MASTs of Q_i and Q_j are merged, while $\{Q_i, Q_j\}$ is a set of two merge expressions which state that the MASTs are not merged. Thus, the most simple merge expression is a single query Q_i. For example, for a set individual queries is $\{Q_1, Q_2, Q_3\}$, $\{Q_{1+2}, Q_3\}$ is one potential merge configuration which holds two merge expressions where Q_1 and Q_2 are merged into one MAST. The problem statement for cost-based merging can thus be re-formulated to find the merge configuration which results in minimal data-redundancy for all queries in the workload W without sacrificing data-locality.

The search space for all merge configuration for n queries in a given workload W is the same as counting the number of non-empty partitions of a set which is defined by the Bell number $B(n)$ as follows [11]:

$$B_n - B_0 = \sum_{k=1}^{n} S(n,k)$$

$S(n,k)$ is the Stirling number of the second kind [11], which counts the number of ways to partition a set of n elements into k nonempty subsets. Our first merge phase reduces the search space, since queries that are contained in other queries are merged (i.e. removed from the workload), which reduces n in the formula above. For example, for TPC-DS we can reduce the MASTs for 99 queries to 17 MASTs (i.e., connected components) after the first merging phase. However, for huge workloads the search space is typically very big after the first merging phase.

Therefore, we use dynamic programming for efficiently finding the optimal materialization configuration for a given workload W. We can use dynamic programming since the optimality principle holds for merge configurations: Let M be an optimal merge configuration for queries $\{Q_1, Q_2, ..., Q_n\}$. Then, every subset M_S of M must be an optimal merge configuration for the queries it contains. To see why this holds, assume that the merge configuration M contains a subset M_S which is not optimal. That is, there exists another merge configuration $M_{S'}$ for the queries contained in M_S with strictly lower data-redundancy. Denote by M' the merge configuration derived by replacing M_S in M by $M_{S'}$. Since M' contains the same queries as M, the data-redundancy of M' is lower than the data-redundancy of M. This contradicts the optimality of M.

We execute dynamic programming to find the optimal merge configuration with n queries. In our dynamic programming algorithm, to find the optimal merge configuration for level l (i.e., with l queries), we execute a binary *merge step* of an optimal merge configuration of level $l-1$ with one individual query. Thus, in total dynamic programming must analyze 2^n different merge configurations. Moreover, a binary merge step must enumerate all possible merge configurations of size l which can be constructed from both

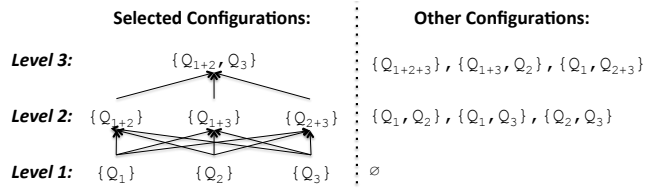


Figure 6: Enumerating Merge Configurations

inputs. Each binary merge step for level l has to analyze maximally l resulting merge configurations. For example, if we want to enumerate all merge configurations of level $l = 4$ which result from merging one merge configuration of level $l = 3$ having two merge expressions $\{Q_{1+2}, Q_3\}$ and a query Q_4, we have to enumerate three resulting merge configurations $\{Q_{1+2}, Q_3, Q_4\}$, $\{Q_{1+2+4}, Q_3\}$, and $\{Q_{1+2}, Q_{3+4}\}$ but not for example $\{Q_{1+2+3+4}\}$. Moreover, memoizing analyzed merge configurations also helps to prune the search space because the same merge configuration might be enumerated by different binary merge steps. For all merge configurations, the binary merge step has to check if the merge configuration is valid (i.e., no cycle occurs in the MAST). Finally, estimating the size for a merge configuration is done by estimating the size for each MAST separately (see Section 3) and then summing up the individual estimated sizes.

Example: Figure 6 shows an example of our dynamic programming algorithm for enumerating merge configurations for three queries. The left hand side shows the selected merge configurations whereas the right hand side shows the other enumerated merge configurations per level. In this example, the optimal merge configuration of the third level $\{Q_{1+2}, Q_3\}$ builds on the optimal merge configuration $\{Q_{1+2}\}$ of the second level.

5. EXPERIMENTAL EVALUATION

In this section we report our experimental evaluation of the techniques presented in our paper. In our experiments we used the TPC-H benchmark (8 tables without skew and 22 queries) as well as the TPC-DS benchmark (24 tables with skew and 99 queries). The goal of the experimental evaluation is to show: (1) the efficiency of parallel query processing over a PREF partitioned database (Section 5.1), (2) the costs of bulk loading a PREF partitioned database (Section 5.2), (3) the effectiveness of our two automatic partitioning design algorithms: schema-driven (SD) and workload-driven (WD) (Section 5.3), and (4) the accuracy of our redundancy estimates and the runtime needed by these algorithms under different sampling rates (Section 5.4).

For actually running queries in parallel, we implemented the PREF partitioning scheme and query processing capabilities over PREF partitioned tables in an open source parallel database called XDB [6].[1] XDB is built as a middleware over single-node database instances running MySQL. The middleware of XDB provides the query compiler which parallelizes SQL queries and then uses a coordinator to execute sub-plans in parallel over multiple MySQL nodes. Thus, the complete query execution is pushed down into MySQL.

[1] https://code.google.com/p/xdb/

5.1 Efficiency of Query Processing

Setup: In this experiment, we have been executing all 22 TPC-H queries on a database with $SF = 10$. We did not use a higher SF since this SF can already show the effects of varying data-locality and data-redundancy. For the experiment, we deployed XDB on an Amazon AWS cluster with 10 EC2 nodes (m1.medium) which represent commodity machines with low computing power. Each m1.medium EC2 node has 1 virtual CPUs (2 ECUs), 3.75 GB of RAM and 420 GB of local instance storage. Each node was running the following software stack: Linux, MySQL 5.6.16, and XDB using Java 8.

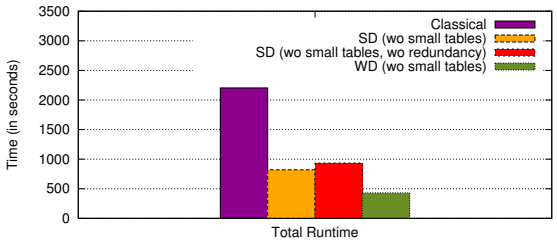


Figure 7: Total runtime of all TPC-H queries

Results: For partitioning the TPC-H database, we compare the following variants where Table 1 shows the resulting data-locality DL and data-redundancy DR for all variants:

- *Classical Partitioning (CP):* This represents the classical partition design in data warehousing [12], where one manually selects the biggest table `LINEITEM` and co-partitions the biggest connected table `ORDERS` to hash partitioned them on their join key. Moreover, all other tables are replicated to all nodes.
- *SD (wo small tables):* This represents our SD algorithm where we remove small tables (i.e., `NATION`, `REGION`, and `SUPPLIER`) from the schema before applying the design algorithm and replicate those tables to all 10 nodes (as discussed in Section 3.1). The SD design algorithm then suggests to use the `LINEITEM` table as seed table.
- *SD (wo data-redundancy, wo small tables):* Compared to the variant before, we additionally disallow data-redundancy for all non-replicated tables. For this variant, the SD design algorithm suggests to use two seed tables (`PART` and `CUSTOMER`) where `LINEITEM` is `PREF` partitioned by `ORDERS`, and `ORDERS` by `CUSTOMER`, while `PARTSUPP` is `PREF` partitioned by `PART`.
- *WD (wo small tables):* Our workload-driven partition design merges all 22 queries into 4 connected components in the first merge phase and then it is reduced to 2 connected components by our second cost-based merge phase: one connected component has 4 tables where `CUSTOMER` is the seed table (while `ORDERS`, `LINEITEM`, and `PART` are `PREF` partitioned) and the other connected component has also 4 tables where `PART` is the seed table (while `PARTSUPP`, `LINEITEM`, and `ORDERS` are `PREF` partitioned).

Figure 7 shows the total runtime of all TPC-H queries. For all variants, we excluded the runtime of queries 13 and 22 since these queries did not finish within 1 hour in MySQL using any of the partitioning configurations (due to expensive remote operations). In fact, when using our optimizations that we presented in Section 2.2, we can rewrite query 13 which uses a left outer join. After rewriting this query finishes in approximately 40s. The total runtime shows that the partitioning configuration suggested by WD (wo small tables) outperforms all other variants. Moreover, both SD variants also outperform CP.

Variant#	DL	DR
Classical	1.0	1.21
SD (wo small tables)	1.0	0.5
SD (wo small tables,wo data-red.)	0.7	0.19
WD (wo small tables)	1.0	1.5

Table 1: Details of TPC-H Queries

For the TPC-H schema, we found that CP represents the best partitioning configuration with minimal total runtime for all queries when not using `PREF`. Thus, CP in this experiment can be seen as an lower bound for existing design algorithms (such as [14, 18]) that are not aware of `PREF`. Consequently, by comparing with CP, we indirectly compare our design algorithms to those algorithms.

Figure 8 shows the runtime of each individual TPC-H query. The results show that whenever a query involves a remote operation the runtime is higher (e.g., the runtime for query 17 and 20 for SD wo redundancy is worse than for SD or WD). Furthermore, when no remote operation is needed but data-redundancy is high in CP, then the query performance also decreases significantly. This can be seen when we compare the runtime of queries 9, 11, 16, 17 for CP with all other schemes. For example, query 9 joins in total 6 tables where 4 are fully replicated and `PARTSUPP` with $8m$ tuples is one of them.

However, when compared to WD which has even a higher total data-redundancy, we see that this has no negative influence on the runtime of queries at all (which seems contradictory to the result for CP). The explanation is that for WD, each query has a separate database (i.e., only the tables needed by that query) which results in a minimal redundancy per query. In fact, the average data-redundancy over all individual databases is even a little lower than for SD (wo small tables). However, when taking the union of all individual databases (of all queries) the data-redundancy is even higher as for CP as shown in Table 1.

Finally, Figure 9 shows the effectiveness of our optimizations that we presented in Section 2.2. Therefore, we execute different queries with (`w`) and without (`wo`) activating these optimizations. As database, we use the TPC-H database $SF = 10$ partitioned using SD (wo small tables). Figure 9 shows the execution time for the following three queries: (1) the first query (left hand side) counts distinct tuples in `CUSTOMER` (which has duplicates), (2) the second query (center) executes a semi join of `CUSTOMER` and `ORDERS` (and counts all customers with orders), and (3) the third query (right hand side) executes an anti join of `CUSTOMER` and `ORDERS` (and counts all customers without orders). The execution times show that with our optimizations the runtime gets efficiently reduced by approximately two orders of magnitude for query (1) and (2). Moreover, query (3) did not finish within 1 hour without optimization while it only took 0.497 seconds to complete with optimization.

5.2 Costs of Bulk Loading

Setup: In this experiment, we bulk loaded the TPC-H

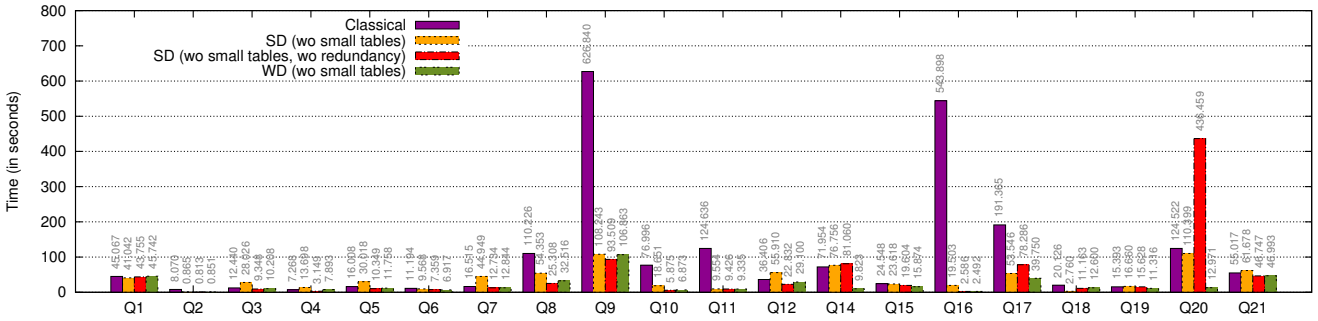


Figure 8: Runtime for individual TPC-H queries

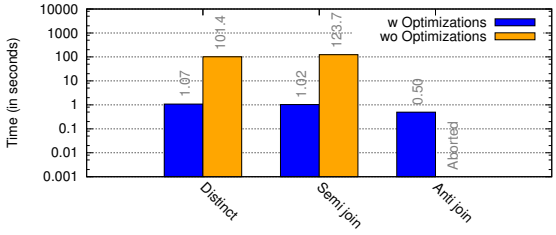


Figure 9: Effectiveness of Optimizations

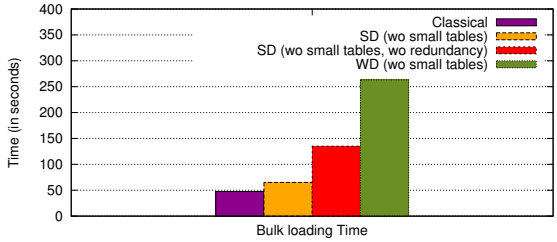


Figure 10: Costs of Bulk Loading

database with $SF = 10$ into XDB. For the cluster setup, we use the same 10 machines as in the experiment before (see Section 5.1).

Results: We report the elapsed time of bulk loading a partitioned TPC-H database for all partitioning schemes discussed in Section 5.1. While the Classical Partitioning (CP) scheme uses only hash partitioning and replication, all other schemes also use `PREF` partitioned tables that are bulk loaded using the procedure described in Section 2.3. Thus, our schemes (SD and WD) have to pay higher costs when inserting a tuple into a `PREF` partitioned table since this requires a look-up operation on the referenced table. However, CP has a much higher data-redundancy (as shown already before) and therefore has higher I/O costs.

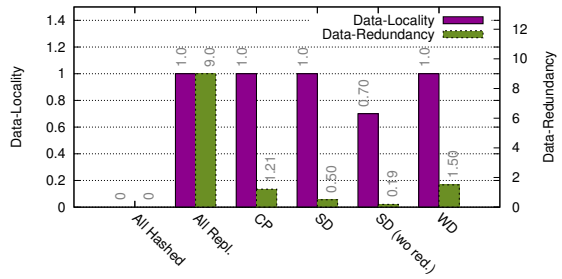
The results in Figure 10 show that the total costs of SD (wo small tables) are only a little higher when compared to CP. In SD (wo small tables, wo redundancy) the costs are a factor 2× higher compared to SD (wo small tables). The reason is that the biggest table `LINEITEM` is `PREF` partitioned where each tuple needs a look-up operation. When disallowing redundancy in SD, it is a common pattern that the biggest table is `PREF` partitioned. The reason is that the biggest table is likely to have outgoing foreign keys that can be leveraged as partitioning predicates without adding redundancy. Finally, WD has the highest bulk loading costs since it pays the costs for higher redundancy and look-ups for bulk loading `PREF` tables. When comparing Figure 7 (Execution Costs) and Figure 10 (Loading Costs), we see that the better query performance is often paid by higher bulk loading costs, which is worthwhile in data warehousing scenarios.

5.3 Effectiveness of Partition Design

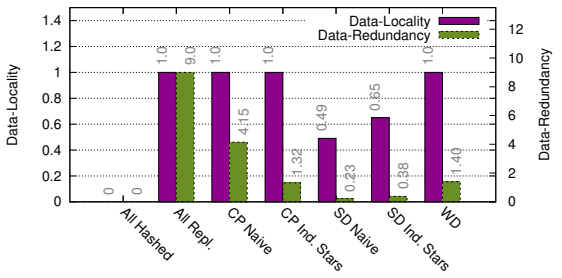
Setup: In this experiment, we use an Amazon EC2 machine of type m2.4xlarge with 8 virtual CPUs (26 ECUs), 68.4 GB of RAM and $2 \cdot 840$ GB of local instance storage to run our partitioning algorithms. The partitioning design algorithms was implemented in Java 8 and we did not parallelize its computation. Compared to the experiments before, we also use the TPC-DS database in this experiment to show the effects of skew.

Results: We first report the actual data-locality and data-redundancy resulting from partitioning a TPC-H and a TPC-DS database of scaling factor $SF = 10$ into 10 partitions (i.e., for 10 nodes). We did not use a higher SF since the results for data-locality and data-redundancy would be very similar for our design algorithms with a higher SF. Afterwards, we show how data-redundancy evolves, if we scale the number of nodes and partitions from 1 to 100 for both databases (using $SF = 10$ for all experiments). This shows how well scale-out scenarios are supported.

TPC-H (10 partitions): For partitioning the TPC-H database, we use all variants shown for the first experiment in Section 5.1. Figure 11(a) shows the data-locality and the actual data-redundancy, which results for the different variants shown before. Additionally, we added to two baselines: *All Replicated* (i.e., all tables are replicated) and *All Hashed* (i.e., all tables are hash partitioned on their primary key). While *All Replicated (AR)* achieves perfect data-locality ($DL = 1$) by full data-redundancy ($DR = 9 = n-1$) where $n = 10$ is the number of nodes, *All Hashed (AH)* has no data-redundancy ($DR = 0$) but at the same time achieves no data-locality ($DL = 0$). Same as *All Replicated*, CP also achieves perfect data-locality ($DL = 1$) with less but still a high data-redundancy. Our design algorithms also achieve high data-locality, however with much less data-redundancy. For example, SD (wo small tables) achieves perfect data-locality ($DL = 1$) with very little data-redundancy ($DR = 0.5$) while WD has a slightly higher data-redundancy ($DR = 1.5$). Moreover, when reducing data-redundancy to $DR = 0.19$ by SD (wo small tables, wo data-redundancy), we still achieve a reasonable data-locality of $DL = 0.7$.



(a) TPC-H



(b) TPC-DS

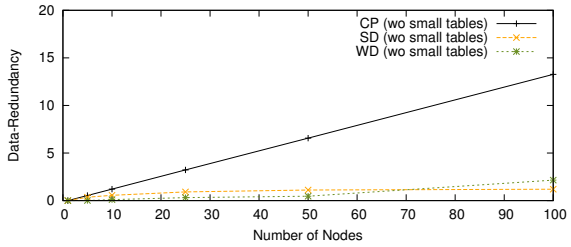
Figure 11: Locality vs. Redundancy

TPC-DS (10 partitions): For partitioning the TPC-DS database, we compare the following variants:

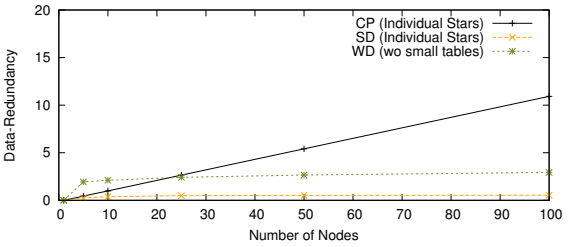
- *CP (Naive and Individual Stars):* This represents the classical partition design as described before. For TPC-DS we applied it in two variants: (Naive) where we only co-partition the biggest table by its connected biggest table and replicate all other tables, and (Individual Stars) where we manually split the TPC-DS schema into individual star schemata by separating each fact table and all its dimension tables into an individual schemata (resulting in duplicate dimension tables at the cut) and then apply CP for each star.
- *SD (Naive and Individual Stars, wo small tables):* For SD we removed 5 small tables (each with less than 1000 tuples) and applied the SD algorithm in the two variants described before: (Naive) where we we apply the SD algorithm to all tables, and (Individual Stars) where we apply SD to each individual star.
- *WD (wo small tables):* We applied our WD algorithm, which merged all 99 queries representing 165 individual connected components (after separating SPJA subqueries) into 17 connected components (i.e., `MAST`s) in the first merge phase and then by dynamic programming we reduced them to 7 connected components (i.e., the number of fact tables).

Figure 11(b) shows the actual data-locality and data-redundancy, which results for the different variants shown before, as well as for the two baselines (*All Replicated* and *All Hashed*). Important is that CP has a higher data-redundancy $DR = 4.15$ to achieve perfect data-locality as for TPC-H. This is due to replicating more tables of the TPC-DS schema. CP (individual stars) involves manual effort but therefore has a much lower data-redundancy $DR = 1.32$. Moreover, while SD introduces even less data-redundancy ($DR = 0.23$), it also achieves a much lower data-locality ($DL = 0.49$) in its naive variant. SD individual stars mitigates this with almost the same DR and $DL = 0.65$. Finally, our WD algorithm results in perfect data-locality (without any manual effort) by adding a little more data-redundancy ($DR = 1.4$) compared to CP (individual stars).

TPC-H and TPC-DS (1-100 partitions): The goal of this experiment is to show the effect of scale-out on the data-locality and data-redundancy of all schemes discussed before. In this experiment, we partition the TPC-H and TPC-DS database of $SF = 10$ into $1-100$ partitions. For partitioning we compare the best SD and the WD variant to the best CP variant of our previous experiments. We do not show both baselines *All Replicated* and *All Hashed*. While for *All Replicated* DR would be linearly growing (i.e., $DR = n$), *All Hashed* always has $DR = 0$. Figure 12 shows the resulting data-redundancy (DR) for TPC-H and TPC-DS: The best CP scheme has a DR which is growing slower than *All Replicated* but has still a linear growth rate. WD and SD have a sub-linear growth rate, which is much lower for big numbers of nodes. Consequently, this means for CP that each individual node has to store more data as for the other schemes when scaling-out. Thus, scaling-out scenarios are not well supported in CP since the performance of query processing will decrease. Note that here we only show data-redundancy, since one can easily reason that data-locality will not change with varying number of nodes for all schemes. Therefore, since the data-redundancy of our approach grows much slower compared to CP, and their data-locality remains unchanged, it means that increasing number of nodes will have a more positive effect on query processing in our approach compared to the CP scheme.



(a) TPC-H



(b) TPC-DS

Figure 12: Varying # of Partitions and Nodes

5.4 Accuracy vs. Efficiency of Partitioning

Setup: We use the same setup as in the experiment before (see Section 5.3).

Results: In this experiment, we show the accuracy of our data-redundancy (DR) estimates when partitioning a TPC-H data-base ($SF = 10$, wo skew) and a TPC-DS database

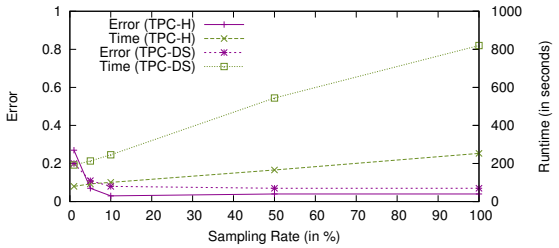


Figure 13: Accuracy vs. Runtime (SD)

($SF = 10$, w skew) for varying sampling rates (i.e., $1 - 100\%$). For showing accuracy, we calculate the approximation error by $|Estimated(DR) - Actual(DR)|/Actual(DR)$. Moreover, we also analyze the runtime effort under different sampling rates (which includes the runtime to build histograms from the database). Figure 13 shows the results of this experiment for the SD (wo small tables) variant. We can see that a small sampling rate of 10% results in a very low approximation error of about 3% for TPC-H and 8% for TPC-DS while the runtime effort is acceptable since it only needs to be executed once (101s for TPC-H and 246s for TPC-DS). The difference in approximation error between TPC-H and TPC-DS can be accounted for by the difference in the data distribution of these two benchmarks. While TPC-H is uniformly distributed, TPC-DS is highly skewed, which results in higher approximation error . The results of WD are not shown in Figure 13) since it has the same approximation error as SD. Moreover, the runtime of WD is dominated by the merge phase which leads to approximately a factor of $10\times$ increase compared to SD.

6. RELATED WORK

Horizontal Partitioning Schemes for Parallel Database Systems: Horizontally co-partition large tables on their join keys was already introduced in the 1990's by parallel database systems such as Gamma [8] and Grace [10] in order to avoid remote join operations. Today co-partitioning is getting even more important for modern parallel data management platforms such as Shark [10] in order to avoid expensive shuffle operations in MapReduce-based execution engines since CPU performance has grown much faster than network bandwidth [19]. However, in complex schemata with many tables, co-partitioning on the join key is limited since only subsets of tables can be co-partitioned which share the same join key. Reference partitioning [9] (or `REF` partitioning for short) is an existing partitioning scheme to co-partition a table by another table referenced by an outgoing foreign key (i.e., referential constraint). Using `REF` partitioning, chains of tables can be co-partitioned based on outgoing foreign keys . However, `REF` partitioning does not support incoming foreign keys. Our `PREF` partitioning generalizes `REF` to use an arbitrary equi-join predicate as partitioning predicate. Another option to achieve a high-data-locality for joins is to fully replicate tables to all nodes in a cluster. However, when fully replicating tables data parallelism typically decreases, since the complete query is routed to one copy and executed locally. Simple Virtual Partitioning (SVP) [4] and Adaptive Virtual Partitioning (AVP) [13] are two techniques that achieve data parallelism for fully replicated databases by splitting a query into sub-queries which read only subsets (i.e., virtual partitions) of the data by adding filter predicates. Compared to `PREF`, for multi-way joins SVP and AVP can only add a predicate to at most two co-partitioned tables which results in expensive full table scans for the other tables in the join path. Moreover, for modern data management platforms with a huge number of (commodity) nodes and large data sets full replication is also not desirable.

Automatic Design Tuning for Parallel Database Systems: While there exists a lot of work in the area of physical design tuning for single node database system, much less work exists for tuning parallel database systems [17, 16, 7, 14, 18]. Especially, for automatically finding optimal partitioning schemes for OLAP workloads, we are only aware of the a few approaches (e.g., those described in [14, 18, 20]). Compared to our automated design algorithms that build on `PREF`, these approaches rely only on existing partitioning schemes (such as hash, range-based, round-robin) as well as replication and decide which tables to co-partition and which to replicate. Moreover, the two approaches in in [14, 18] are tightly coupled with the database optimizer. However, in this paper we show that our partitioning design algorithms, which are independent from any database optimizer, can give a much better query performance by efficiently using on our novel `PREF` scheme (even when not knowing the workload in advance). Recently, different automatic design partitioning algorithms have been suggested for OLTP workloads [17, 16, 7]. However, the goal of these approaches is to cluster all data used by individual transactions on a single node in order to avoid distributed transactions. For OLAP, however, it is desirable to distribute the data needed for one transaction (i.e., an analytical query) evenly to different nodes to allow data parallel processing. Thus, many of the automatic design partitioning algorithms for OLTP are not applicable for OLAP workloads.

7. CONCLUSIONS AND OUTLOOK

In this paper, we presented `PREF`, a novel horizontal partitioning scheme, that allows to co-partition a set of tables by a given set of join predicates by introducing duplicates. Furthermore, based on `PREF`, we also discussed two automatic partitioning design algorithms that maximize data-locality while minimizing data-redundancy. While our schema-driven design algorithm uses only a schema as input and derives potential join predicates from the schema, the workload-driven algorithm additionally uses a set of queries as input. Our experiments show, that while the schema-driven algorithms works reasonably well for small schemata, the workload-driven design algorithm is more efficient for complex schemata with a bigger number of tables.

One potential avenue of future work is to adopt our automatic partitioning design algorithms to consider data-locality also for other operations than joins only (e.g., aggregations). Moreover, it would also be interesting to adopt our partitioning design algorithms to dynamic data (i.e., updates) and for mixed workloads (OLTP and OLAP) as well as for pure OLTP workloads. We believe that our partitioning design algorithms can also be used to partition schemata for OLTP workloads (when we disallow data-redundancy for all tables) since tuples that are used by a transaction can typically be described by a set of join predicates. Finally, partition pruning for `PREF` is another interesting avenue of future work.

8. REFERENCES

[1] Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[2] TPC-DS. http://www.tpc.org/tpcds/.

[3] TPC-H. http://www.tpc.org/tpch/.

[4] F. Akal, K. Böhm, and H.-J. Schek. OLAP Query Evaluation in a Database Cluster: A Performance Study on Intra-Query Parallelism. In *ADBIS*, pages 218–231, 2002.

[5] C. Binnig, N. May, and T. Mindnich. SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. In *BTW*, pages 363–382, 2013.

[6] C. Binnig, A. Salama, A. C. Müller, E. Zamanian, H. Kornmayer, and S. Lising. XDB: a novel database architecture for data analytics as a service. In *SoCC*, page 39, 2013.

[7] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[8] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[9] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in Oracle. In *SIGMOD Conference*, pages 1111–1122, 2008.

[10] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB*, pages 209–219, 1986.

[11] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[12] H. Herodotou, N. Borisov, and S. Babu. Query optimization techniques for partitioned tables. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 49–60, 2011.

[13] A. A. B. Lima, M. Mattoso, and P. Valduriez. Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster. *JIDM*, 1(1):75–88, 2010.

[14] R. V. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD Conference*, pages 1137–1148, 2011.

[15] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[16] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD Conference*, pages 61–72, 2012.

[17] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *EDBT*, pages 430–441, 2013.

[18] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In *SIGMOD Conference*, pages 558–569, 2002.

[19] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. In *ICDE*, 2014.

[20] T. Stöhr, H. Märtens, and E. Rahm. Multi-Dimensional Database Allocation for Parallel Data Warehouses. In *VLDB*, pages 273–284, 2000.

[21] F. M. Waas. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *BIRTE (Informal Proceedings)*, 2008.

[22] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[23] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD Conference*, pages 13–24, 2013.

APPENDIX

A. ESTIMATING REDUNDANCY

As discussed in Section 2, predicate-based reference partitioning might result in some redundancy in `PREF` partitioned tables. Since seed tables in our partitioning design algorithms will never contain duplicate tuples (and thus redundancy), in order to estimate the size of a database after partitioning, the expected redundancy in all tables which are `PREF` partitioned must be estimated. In the following, we explain our probabilistic method to estimate the size of a given `PREF` partitioned table.

Redundancy is cumulative, meaning that if a referenced table in the `PREF` scheme contains duplicates, the referencing table will inherit those duplicates as well. This can be best explained by the example in Figure 2. Table `ORDERS` has two copies of the tuple with `orderkey=1`. Table `CUSTOMER`, which is `PREF` partitioned by `ORDERS` inherits this duplicate (i.e., customer with`custkey=1` is therefore stored redundantly in partition 1 and 3). In other words, `PREF` partitioning can be viewed as walking a tree, where nodes are the tables (the seed table being the root) and each referenced table is the parent of its referencing table(s). The redundancy in the parent table results in the redundancy in the child table. Therefore, in order to estimate the redundancy in a `PREF` table, we should take into account the redundancy of all the tables along the path to the seed table.

To this end, we assign a *redundancy factor* (denoted by r) to each edge in the `MAST`. To find the redundancy factor of an edge, we use histogram of the join key in the referenced table (whereas we can use sampling to reduce the runtime effort to build histograms). For example, assume that we want to estimate the size of the table `ORDERS` in Figure 2 after partitioning. For each value in column `orderkey` of table `LINEITEM`, we calculate the expected number of duplicates of the corresponding tuple in table `ORDERS` we expect after partitioning (i.e. the number of partitions which contain a copy of that tuple). The idea behind this method is that tuples with lower frequency in the histogram are expected to end up in fewer number of partitions (i.e. fewer duplicates) as compared to tuples with higher frequency. Therefore, by calculating the *expected value* of copies of each distinct value in the join key of the referenced table, and then add them all together, we can find an estimate of the size of the referencing table after partitioning.

We now formally explain our method. Let the random variable X denote the expected number of copies of a tuple after partitioning, n represent the number of partitions and f denotes the frequency of that tuple in the histogram. Note that X can take any number between 1 and $m = min(n, f)$: $X = 1$ results when all references of that tuple happen to be in the same partition, and therefore, no replica is needed. On the other hand, $X = min(N, f)$ is the maximum value, since the number of copies of a tuple is either n (i.e. full replication) or f, when $f < n$ and each reference ends up in a separate partition. The expected number of copies of a tuple with frequency f is therefore as follows:

$$E_{f,n}[X] = 1 \cdot P_{f,n}(X = 1) + 2 \cdot P_{f,n}(X = 2) + \cdots + m \cdot P_{f,n}(X = m)$$

where $P_{f,N}(X = x)$ is the probability that the tuple has x copies (meaning that it will be replicated to x different

partitions, out of the total N partitions). It can be calculated as follows:

$$P_{f,n}(X = x) = \frac{\binom{n}{x} \cdot x! \cdot S(f,x)}{n^f}$$

In the formula above, $S(f,x)$ is the *Stirling number of the second kind*, and is the number of ways to partition a set of x distinguishable objects (tuples) into n non-empty indistinguishable boxes (partitions). The numerator, therefore, is the number of ways to choose x partitions out of total n partitions, i.e. $\binom{n}{x}$, and then to partition f tuples into these x distinguishable partitions, which is why the Stirling number is multiplied by $x!$. The denominator is the total number of ways to put f distinct tuples into n distinct partitions.

The above mentioned formula requires a number of expensive recursions (because of the Stirling number). Since $E_{f,n}[X]$ depends only on f and n, the entire computation can be done in a preprocessing phase. Therefore, instead of actually calculating the expected number of copies for each tuple in run-time, only a fast look-up in a pre-loaded table is enough. Thus, the time complexity of finding $E_{f,n}[X]$ in run-time is $O(1)$.

We are now ready to calculate the *redundancy factor* of an edge in `MAST`. We define it as the after-partitioning size of a table divided by its original size. Let V_e denote the set of distinct values in the join key of the referenced table T_i over its outgoing edge e to table T_j(for example,the distinct values of column `orderkey` of table `LINEITEM` in Figure 2 is $\{1,2,3\}$).

The *redundancy factor* is defined as follows:

$$r(e) = \frac{\sum_{v \in V_e} E[v]}{|T_j|}$$

Note that $r(e)$ can be ranged between two extremes: 1 (no redundancy) and n (full redundancy). As mentioned before, the after-partitioning size of a referencing table is not determined only by the redundancy factor of the immediate edge coming from its referenced table, but also by the redundancy factor of all the edges along the path from the seed table.

Finally, we can now estimate the after-partitioning size of table T_i:

$$|T_i^P| = |T_i| \cdot \prod_{e \in path(T_{RF}, T_i)} r(e)$$

where $path(T_{RF}, T_i)$ consists of all the edges from T_{RF} to T_i in the `MAST`.

For example, assume that $r(\texttt{LINEITEM} \rightarrow \texttt{ORDERS})$ turns out to be 2, meaning that $|\texttt{ORDERS}^P|$ would be twice as big as its original size $|\texttt{ORDERS}|$. Now, if $r(\texttt{ORDERS} \rightarrow \texttt{CUSTOMER})$ is 3, the after-partitioning size of table `CUSTOMER` would be estimated to be $2 \cdot 3 = 6$ times its original size.