Partitioning Key Selection for a Shared-Nothing Parallel Database System

Appears as *IBM Research Report RC 19820 (87739) 11/10/94*, IBM Research Division, T. J. Watson

Reaesrch Center, Yorktown Heights, NY, Nov. 1994.

Daniel C. Zilio
Department of Computer Science, University of Toronto

Anant Jhingran
Sriram Padmanabhan
IBM TJ Watson Research Center,
Yorktown Heights, NY

Abstract

A shared nothing database system which tries to leverage the knowledge of partitioning attributes of relations can outperform a system where such knowledge is either not available or not used. The performance improvements are typically obtained by function shipping more database operations (joins, aggregates etc.), thus minimizing the communication overhead. In such a system, it is critical that the correct partitioning keys are selected so that the query workload is optimized. Previous research has ignored the importance of selecting the partitioning keys and have mostly focused on the degree of declustering. In this study we show that by following a systematic methodology, especially for the partitioning key selection and associated relation grouping issues, the entire data placement strategy for a given database schema and workload can be determined in a very efficient manner. We describe different flavors of this methodology and demonstrate the performance improvements resulting from them.

1 Introduction

It has been seen that in order to achieve the performance required for current and future decision support system (DSS) and on-line transaction processing (OLTP) applications, parallel database systems should be used [DG92]. Products announced from Teradata [FK92], IBM [DBPE94, Perna94], Tandem [CC93], Oracle [Linder93], Informix [Clay93], etc., as well as research projects such as Bubba [B+90] and GAMMA [D+90] all attest to this fact. Most parallel database systems require some form of *Data Placement* in order to partition the I/O and CPU workload among various processors thereby improving the response time and throughput of the workload.

The current generation of Decision-Support applications already require data capacities of hundreds and thousands of GigaBytes. Besides high capacity and complex query performance requirements, these applications also require scalability of both the data and the workload. It is widely recognized that the *Shared-Nothing* database architecture is most suited for such requirements [DG92]. By virtue of the underlying hardware, data must be partitioned and distributed across the different nodes in the system and a *function shipping* model of execution is required for query execution. In the function shipping model, the data is accessed and processed at the node where it resides in order to minimize communication between nodes. Wherever possible, predicate evaluation,aggregation, sorts and joins are done locally. In turn, this requires that the data be initially placed on the system so as to facilitate such parallel optimizations. Thus, the *Data Placement* issue is very important in Shared-Nothing parallel databases.

We use the term **Data Placement** to refer to the specific *parallel database layout* issues that must be addressed as part of the database design. Data placement should facilitate parallelism and minimize the communication overheads of processing in order to improve the overall performance of the database workload. The reasons for trying to minimize communication are: (a) It adds to the response time of the queries, (b) It utilizes precious CPU cycles (decision support applications are often CPU intensive) and (c) network, being the *only* shared resource, can saturate when excessive data transfers are performed.

Excess communication is likely to happen in those database systems where data placement is not under user control (generally, round-robin or random), or where the optimizer and run-time operators do not recognize any specific partitioning attributes. For example, consider the following query: `"select * from t1, t2 where t1.a = t2.a"`. In the case of database systems such as GAMMA [D+90], the only join strategy would be to *repartition* both t1 and t2 on a new set of nodes, and then perform the parallel join. In contrast, in a system such as DB2 Parallel Edition [DBPE94], if t1 and t2 are partitioned on t1.a and t2.a respectively, then the entire communication cost of repartitioning both the relations can be avoided. In this paper we assume that the database software is able to optimize queries to use the database partitioning.

However, communication can sometimes not be avoided even if the database software recognizes the partitioning attributes. If the query were `"select * from t1, t2 where t1.b = t2.b"`, then the above partitioning attributes would still require a repartitioning step, as in GAMMA. It is this problem that we address in this paper: what is the correct choice of partitioning attributes (and the other associated data placement issues) so that the overall query workload is optimized?

There are three aspects to data placement (and hence parallel database layout design) [Padman92]:

1. Choosing the *partitioning attributes* for the relations. Thus, in the examples above, we would like to determine whether to partition t1 on t1.a or t1.b. We would also like to *group relations* so that they share a "declustering strategy". If we want to localize the joins between t1 and t2, not only do they have to be partitioned on the joining keys, but *every matching tuple pair must be on the same node.* In this paper, we assume that this condition is guaranteed if the two relations belong to the same relation group.

2. Determine the *degree of declustering* for the grouped relations.

3. Determining the actual *assignment* for a group, i.e., the actual node numbers for the relations in the database.

Past work has pointed out that data placement is necessary on the shared nothing systems [CABK88, GDa90, GDb90, D+90, LKB87, PB92, Padman92, MD94]. However, all the past work have focused on the declustering and assignment aspects of data placement and have paid minimal attention to the details of partition attribute selection. Some of the past studies [CABK88, GDb90, Padman92], have mentioned the importance of partition attributes but have not paid proper attention to the subject. In fact, they assume that the proper partitioning attributes have been chosen as a pre-requisite for their declustering and assignment steps.

We believe that this paper is the first to study the partitioning attribute selection problem, and the related and important problem of relation grouping for locality of access in the context of an integrated data placement methodology. We present integrated data placement algorithms that, given the database schema and the workload characteristics, generate *an optimal partitioning strategy* and follow that up with an efficient declustering and assignment of relations so that the overall response time of the workload is minimized. However, we discuss the second and third aspects only in passing and refer the reader to prior art.

The determination of the optimal partitioning key for each relation is complicated by (a) database schemas involving complex relationships, (b) workload characterization being imprecise or too dynamic and (c) insufficient information on data distribution on various candidate partitioning keys (thus making a balanced data load difficult). However, we believe that even where decision support applications submit ad hoc queries, the joins between relations happen in pre-specified ways and the database designer has a fairly good idea of the key operations on the query. Furthermore, most decision support applications have fairly static data characteristics (i.e., data does not change rapidly) – making the information on skew, etc., easy to obtain and maintain. Of course, it makes sense to rerun the the data placement algorithm if the database schema, the workload characteristics and the data distribution have changed significantly. This paper does not address the parallel reorganization issues introduced by such dynamically changing data and workload descriptions and only focuses on the initial data placement issue.

In this paper, we will assume a static database schema, a given workload (actual queries and their relative frequencies), information on data skews for various attributes, and the overall system size (number of nodes etc.). The more precise the information is, the more likely we are to be correct in the data placement output. However, the central thesis of this paper is to show that given a set of inputs, the placement algorithms can relatively easily identify the most appropriate partitioning keys of relations, and that this leads to significant performance benefit. We will also show that for a little more effort, we can tweak our partition keys for slightly better performance and that there is almost never a need for an exhaustive exponential search through the candidate key space.

Our placement algorithms determine a few candidate partitioning keys for each relation using *attribute weighting* wherein each possible partition key is assigned a weight based on its usage in the workload. The first placement algorithm, called *Independent Relations*, makes decisions for relations independently when choosing the partitioning attributes (i.e. it considers what happens when we choose t1.a as the partitioning key). It then considers the optimal grouping for its chosen partitioning keys. The second algorithm, called *Comb*, considers different combinations of keys across relations (i.e. what happens when we choose t1.a *and* t2.b as partitioning keys) and combines it with the relation grouping step. Both the algorithms use a simple declustering and assignment methodology, similar to Mehta and De Witt's algorithms [MD94] as their final steps. The important focus in our algorithm is the partitioning attribute and grouping algorithms, so other declustering and assignment algorithms could generally replace the current ones in the future if required.

The rest of the paper is structured as follows. Section 2 will present a motivation of the data placement problem with particular focus on the partitioning issues. With this in mind, we present different general schemes for partitioning key selection and placement in Section 3. This section illustrates our simplifying assumptions and the usefulness of our algorithm compared to other approaches. A detailed description of our placement algorithms, with special focus on the partitioning and relation grouping steps, is presented in Section 4. Section 5 describes performance comparison experiments based on an analytical model to show the effect of good placement on workload performance. Finally, section 6 summarizes the important contributions of the paper and describes topics of future interest.

2 Motivation

As mentioned above in the introduction, the goal of parallel database placement involves the steps of (i) choosing partitioning keys for relations and grouping relations, (ii) determining declustering degrees for the relation groups, and (iii) assigning these relations to nodes. Of these, step (i) is the important focus of this paper. The declustering and assignment steps are also integrated into the complete data placement methodology.

The various forms of partitioning include: random partitioning; round-robin partitioning; hash partitioning; and range partitioning. Shared-Disk and some small scale Shared-Memory systems are examples of systems that do not care about partitioning attributes. They randomly distribute tuples across the disks in order to balance the data in each of the relation's partitions, e.g., round-robin or random partitioning. However, if a Shared-Nothing system employs one of these techniques, then

```
Customers(C_CUSTKEY,C_NATION,....)
Orders(O_ORDERKEY,O_CUSTKEY,O_ORDERDATE,...)
Lineitem(L_ORDERKEY,L_SUPPKEY,L_EXTENDEDPRICE,L_DISCOUNT,...)

SELECT L_ORDERKEY, O_ORDERDATE, ...
FROM Customers, Orders, Lineitem
WHERE C_MKTSEGMENT = [segment]
AND C_CUSTKEY = O_CUSTKEY
AND L_ORDERKEY = O_ORDERKEY
AND ...
GROUP BY L_ORDERKEY, O_ORDERDATE,...
```

Figure 1: Simplified Relation Schema and query from the TPC-D benchmark specification.

it can only help in obtaining improvements from data balancing and application of data predicates in parallel. However, other SQL operations like joins, aggregates etc. cannot be fully localized. In order to obtain these benefits, hash or range partitioning strategies should be employed. In hash and range partitioning, one or more attributes serve as partitioning attributes to determine buckets or ranges on which the relation will be partitioned.

Performance gains can be achieved when partitioning attributes are selected on the basis of their use in queries. An example of such a gain can be shown in the execution of a join operator which we will demonstrate using a query example from the evolving TPC-D [Raab94] industry standard benchmark. Figure 1 shows the schema of the relations involved in the query and the query description. The query performs joins on the `Customers` , `Orders` , and `Lineitem` relations. If `Customers` were partitioned on C_CUSTKEY and `Orders` on O_CUSTKEY, then the join operation between them can be performed *only* using local partitions. If they were not the partition keys, then this join would require *repartitioning* of one or both relations for the best case execution (e.g., hash-join algorithm [DG92]). Similarly, O_ORDERKEY and L_ORDERKEY choices would enable the join between `Orders` and `Lineitem` to be performed locally. Note, that there are now two partitioning candidates for the ORDERS relation, O_CUSTKEY and O_ORDERKEY. Choosing between the two is dependent on the importance of each join. Since the same situation can occur on other queries, we could have several candidates for each relation. Choosing the partition attributes from this set manually could lead to incorrect choices and poor performance. Our partition key selection algorithms are designed to be able to select a good partitioning candidate for each relation.

In fact, almost every SQL construct (aggregation, subqueries, inserts, updates, deletes, etc.) benefits from a favorable partitioning of the relations. An experimental comparison of one of the partition key selection algorithms proposed in this paper with a round-robin (or random) partitioning strategy using the methodology described in Section 5 showed that use of partitioning keys improved the response time of this query by 30%. Lack of proper partitioning (equivalently, database system not being cognizant of partitioning) can result in excessive repartitioning, resulting in higher CPU and communication costs. A point in decision support queries is often overlooked – they are highly CPU intensive; consequently, communication penalizes the system not only due to contention of shared resource, but also due to increased CPU utilization for processing all the messages.

While an attribute could appear as a good candidate from one query's perspective it could

- lead to data imbalance among nodes, either because the number of unique values in its active domain is too low (e.g., only Male/Female values for the *Sex* attribute) or because of a high skew in its values distribution.
- benefit the performance for a few queries, at the expense of others, and the benefits are outweighed by the penalties.

Thus, there are a lot of factors and inputs involved in choosing the correct partitioning attribute set. Focusing on the usage of attributes in characteristic queries is important but factors such as the relation schema, column cardinality, and skew information are also useful. Therefore, a good data placement algorithm algorithm requires the workload and database inputs to make a proper partitioning attribute decision.

Because of all the factors and inputs involved, the data placement algorithm decision is not a trivial one. For example, choosing whether to partition `Orders` on join attribute O_CUSTKEY or join attribute O_ORDERKEY is not an easy problem. The attribute selection depends on the number of occurrences in joins on `Orders` , their cardinalities and skew, and the frequency of the joins with respect to other operations on the relation.

Besides choosing the partitioning attributes, the data placement algorithm will choose the *relation grouping.* A Relation Group is a logically connected group of relations which can benefit from locality if they are considered together for declustering and assignment. It uniquely identifies a *declustering strategy* that given a partition key, determines the specific node where it belongs. All relations in a group can consequently be joined without repartitioning (provided the joins are on their respective partitioning keys). By corollary, relations of a group have identical the degree of declustering and node assignments. DB2 Parallel Edition [DBPE94] provides such a flexibility using the concept of *Node Groups.* In addition, DB2 Parallel Edition can load balance different groups differently by using different mapping of partitioning key to node numbers.

To give an example of node grouping, consider three tables t1, t2 and t3 belonging to two relation groups rg1 and rg2 (t1 and t2 in rg1 and t3 in rg2). Let the partitioning key of each be a. Then a given value of t1.a is guaranteed to be on the same node as t2.a, but no guarantee is made about t3.a. If range partitioning is used, the range to node assignment of t1 and t2 could be different from t3. We see that additional flexibility is obtained in declustering, node assignment and load balancing when tables do not belong to the same node group. However, the flip side is that some joins are no longer local (e.g., a join between t1.a and t3.a). We would therefore like to assign to a common group only those relations where the benefits of local join processing likely outweigh the lack of flexibility.

Finally, we need to make sure that the data is well balanced among all the nodes. The decision depends on the system configuration and the database usage requirements; if throughput is the sole criteria, the degree of declustering for each relation need not be the largest possible. However, assuming that the criteria are mainly response time improvements, the declustering for relation groups containing large relations is fairly obvious (assuming that system configuration is based on capacity) – all the nodes. However for smaller relations, we want to balance the load across processors. Thus a third output of the data placement algorithm is the actual node assignment for each relation group.

Now it can be easily argued that these three steps are interdependent. For example, if we do decide not to group two relations together, does it make sense to partition both on their joining attribute? (Recall that if they are not grouped together, locality of join processing can no longer be guaranteed.) Similar examples for other interactions can be constructed. However, we show in this paper that viewing these three as separate steps is not only efficient but also leads to a relatively optimal solution for data placement.

3 Possible Data Placement Algorithms

The structure of a general data placement algorithm is given in figure 2. The inputs to the algorithm will consist of the database schema (DDL), system information (e.g., number of nodes, N), workload characteristics, and some important statistics. Workload characteristics consists of the query structures, Q_i, including operations and the attributes they require which are filtered from the SQL statement for the query, and the query frequencies, f_i. The statistics give the data placement algorithm added information to help make its decisions. Statistics useful to the algorithm include relation and column cardinalities, and skew information of columns. The algorithm will output the data placement, which includes the set of partitioning attributes, $\{Pkey\}$, the relation groups, RG_j, and the declustering of the relations across the system.

Like other optimization problems, any data placement optimization must balance its cost with respect to the benefit it provides. It does not make sense for an algorithm to have high execution times if its benefit is not high or for it to have low cost if the benefit is minimal. For data placement, the two extremes are:

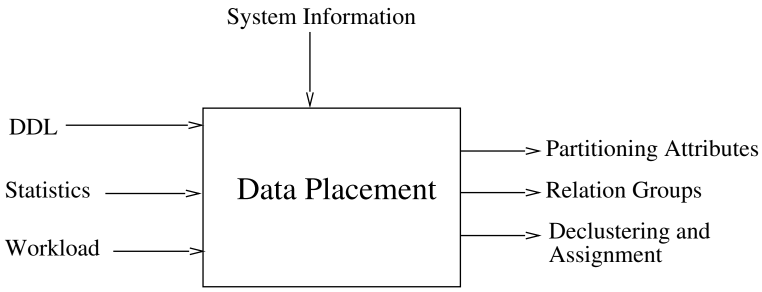

Figure 2: Generic data placement algorithm

- *Exhaustive Search*: It should be obvious that an algorithm which looks at all possible partition key choices, relation groupings and node assignments would be provably optimal – however, its execution cost could be so high as to render it impractical.

  One major problem with the exhaustive algorithm is that there are a lot of possible data placements. Let n_i be the number of attributes in relation i, and k be the number of relations. The total number of partitioning attribute sets is $O(2^{\Sigma_{i=1}^{k} n_i})$. If $k = 5$ and $n_i = 5$, this number is at least 3×10^7.

  Another problem with trying to come up with an optimal strategy is: how exactly do we evaluate the various partitioning attributes and potentially the groupings and node assignments? Clearly the cost of one data placement must be determined efficiently – this rules out simulation or actual database execution using a candidate data placement. There are two methods of choice, (i) analytical methods, or (ii) query optimization based iteration. The analytic method, while likely to be efficient, tends to ignore the actual join sequence, methods etc., hence it does not predict the cost correctly. The optimizer, on the other hand does consider join sequences and methods but suffers from the following drawbacks: (i) It is certainly not time-efficient, especially if there are 30 Million iterations! (ii) Its response time predictions are only approximate at best (thus even using it cannot guarantee optimality).

- *Random Selection*: We could randomly pick a partitioning key for each relation, assign each relation to a separate group, and assign all nodes to that group. This random selection is extremely fast but the actual performance of the queries based on this data placement algorithm algorithm may be extremely suboptimal.

Of course, we could have lots of options in between. The random selection could be extended to eliminate high skew attributes, choose degree of declustering based on size of the relation etc. and hence improve the workload performance. The exhaustive search could use the same heuristics and cut down its complexity. In this paper, we present two algorithms that have complexity between the two extremes described above.

The two algorithms, *Independent Relations (IR)* and *Combined Relations (Comb)* share the following properties:

- Elimination of "bad" attributes – attributes that have high skew, low column cardinality, etc.
- Assigning weights to individual partition keys and eliminating those that do not meet a certain threshold and choosing a partitioning key from the remaining.
- Grouping of relations as a separate step after partition key selection.
- Assignment of nodes to a group as a separate and final step.

The justification of separation of the declustering and assignment steps from other steps has been given in a lot of previous studies [CABK88, Padman92, PB92, GDa90, GDb90, Ghand90, MD94].

The basic difference between the two algorithms is in the selection of partitioning keys – IR chooses one randomly among candidates assigned the highest weight, whereas *Comb* pares down the partition key candidates using their weights and exhaustively considers combinations of keys across all relations before choosing the best combination.

4 IR and *Comb* Algorithms

We first describe the IR algorithm and later describe the important modifications in the *Comb* algorithm.

4.1 The IR algorithm

Figure 3 describes the main steps of the IR data placement algorithm. There is an initial *filtering step* which eliminates unfit attributes of each relation and certain small relations. The next step is the *Attribute-Weight* based partition key selection algorithm. In this step, we consider the usages of different attributes of each relation in certain important operations in the workload queries and compute their weights. The highest-weighted attribute is chosen as the partition key of a relation in the IR algorithm at the end of this step. The next step is *relation grouping*, where the relations are grouped together based on their partition keys. This is followed by the final step of *declustering and assignment*. Each of these steps are explained in greater detail in the following subsections.

4.1.1 Attribute and Relation Filtering Step

In either hash-based or range partitioning, low column cardinality or skew in data distribution can lead to imbalanced load. For example, we can use the following empirical rule to eliminate bad attribute candidates. If the number of distinct values in an attribute $< 10N$, where N is the number

```
Independent_Relations Algorithm
-------------------------------
    /* Inputs:
       Workload - queries and frequencies
       DDL - relation schemas and number of relations
       Statistics - e.g., cardinality information
       Sysinfo - system information  */

    1. Eliminate Bad Attributes which have high skew
       or low cardinality based on statistical input.
       Eliminate small relations from next step.

    2. Partition Key Selection:
       Calculate aggregate weights for  attributes
       of all relations by traversing all the
       queries in the workload.

       Choose highest weighted attribute as partition
       key of each relation. Break ties using cardinality
       of columns.

    3. Relation Grouping:
       Perform transitive closure of selected partition keys
       based on join clauses appearing in queries.

    4. Declustering and Assignment step.
       Assign large relations to all nodes.
       Assign all small un grouped relations with
       data balancing as criterion.
```

Figure 3: The Independent Relations Algorithm

of system nodes, or if some particular value(s) occur in more than $0.5/N$ tuples, the probability is high that good data distribution will not be obtained using that key.

In this step, we also mark some relations as not requiring a specific partitioning key. Relations which are really small are either to be partitioned on one node, or they typically join other relations by being broadcast to the set of nodes containing the other relation. In both these cases, the choice of partition key is irrelevant. However, as part of the relation group step, a small relation is allowed to be grouped with a larger relation depending on the joins between them. The cost saving in localizing a join with the larger relation may sometimes offset the overhead of spreading the smaller relation too thinly. If no such join exists, the small relation is put into an independent relation group and placed by the assignment algorithm described in section 4.1.4.

4.1.2 Partition Key Selection

In this important step, we use *pre-assigned weights* for the various operators that are important from the standpoint of parallel query optimization. If an attribute of a relation participates in one of these operations in a query then its aggregate weight is incremented or decremented by the assigned weight. The final aggregate weight of an attribute is based on all the queries in the workload and it reflects the importance of the attribute in the context of the workload. The justification of this approach is as follows. If the attribute is chosen to be the partitioning key of the relation, then it is likely to benefit the set of operations which contribute to its aggregate weight. On the other hand, if any other attribute is chosen then all these operations most likely will require redistribution of data. Thus, our method of aggregating the weights and choosing the highest weighted attribute captures the benefit of choosing partitioning attributes of relations most effectively.

As mentioned before, almost all SQL operations can benefit from compatible partitioning. However, the following are the most relevant in our study – equi-joins, Group By operations, duplicate elimination, and selections. We describe each in detail, along with their relative importance:

- Joins: An equi-join would benefit greatly if one or more of its join attributes were chosen as the partitioning attributes. The reduction in the communication cost for an equi-join will be proportional to the number of tuples of one of the relations or the sum of the number of tuples for both relations. This indicates that the savings is substantial when considering join attribute(s) as partitioning attribute(s) and when the relations have high cardinalities. We assign joins the highest importance, giving them a weight of 1.

- Group By: A query could benefit from localizing the grouping of its tuples at each node based on the columns involved in the *GROUP BY* clause. A *GROUP BY* clause will cause tuples to be aggregated together yielding a tuple for each distinct set of values for the attributes listed in the clause. If the Group BY operation cannot be performed locally, it would require global coordination across all the nodes of the intermediate result. The reduction in cost here is proportional to the number of groups and using a rule of thumb of one group per ten tuples, a weight of 0.1 is assigned to a grouping attribute.

- Distinct Elimination: If the attribute(s) are partitioning attributes, the duplicate removal can be localized, i.e., the sorting and removal of duplicates can be local to each node. The Distinct operation is slightly less expensive than the Group By operation in general. Also, it is one of the last operations of a query or subquery, so it mostly operates on a smaller set of tuples. For these reasons, we allocate a weight of 0.08 for duplicate elimination.

- Selections: Equality selections or exact matches, e.g., $R.A = 5$, could help decide whether the selected attribute is to be rejected or considered as the partitioning attribute. An exact

	Operator	Weighting Factor
1	Join	1.0
2	Group By	0.1
3	Duplicate Removal	0.08
4	Selection (constant)	-0.05
5	Selection (Host variable)	0.05

Table 1: Relative weightings between database operations

match of an attribute to a constant causes only one partition to be required for the selection operation if that attribute was chosen as the partitioning attribute. Thus, at least part of the query's execution would always be localized to one specific node resulting in load imbalance. If the selection was against an input host variable, which could have different values on different invocations of the query or transaction, partitioning on this attribute could help in allowing the queries to be executed on different nodes in parallel. An example of such a variable is [*mktsegment*] in the TPC-D example of figure 1. Anyway, being able to localize execution is good from a throughput point of view; not necessarily response time, and hence this usage is assigned a weight of 0.05.

The relative weights of the attributes are summarized in Table 1. We have found that the relative difference between these values is sufficient to bring out the differences between the operations and clauses as likely indicators for partitioning key selection.

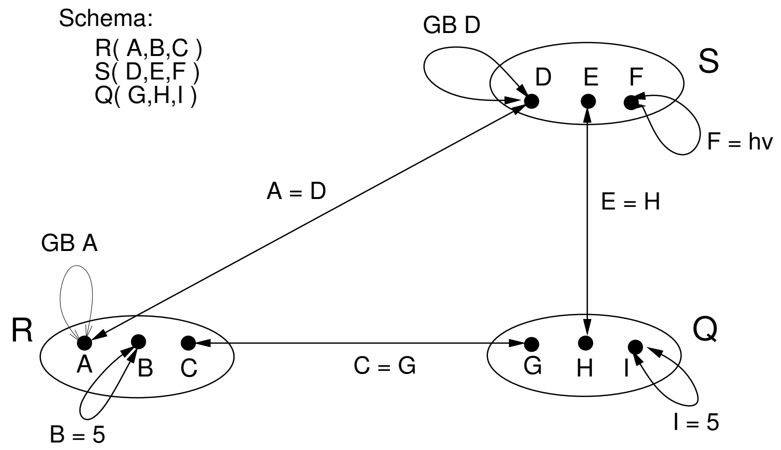

Figure 4: Example of a query graph on three relations R, S, and Q. Edges represent query operations.

During this phase, the workload is traversed and weights are assigned to each attribute according

Relation	Attribute	Weight
R	A	1.10
	B	-0.05
	C	1.00
S	D	1.10
	E	1.00
	F	0.05
Q	G	1.00
	H	1.00
	I	-0.05

Table 2: Aggregate Weight of attributes for relations R, S, Q from previous workload.

to its occurrence. That is, if an attribute $R_i.A_j$ occurs in the workload query Q_k with a frequency f_k in an operation m (where $1 <= m <= 5$, refer to Table 1), then the weight of $R_i.A_j$ is increased by the product $f_k W_m$. We illustrate the Attribute Weight algorithm using the following example. Consider the schema and workload shown in figure 4. In the figure, joins are represented by edges between relation vertices in the graph. Exact matches on attributes B and C are represented by edges onto the same attribute. Note that GB is a short-form for *GROUP BY*, and hv represents a variable that could have different values on different invocations of the query. All the edges in the graph are traversed and the appropriate weights are aggregated on the nodes. The result of the weight assignments is given in Table 2.

After the queries have all been traversed, the partitioning attributes chosen are the ones with the highest weight in each relation. In the case of ties, the attribute(s) with the higher cardinality or lowest skew would be the winner(s). If this extra cardinality information is not given by the user, the choice among the ties is arbitrary. Thus, $R.A$, $S.D$ and one of $Q.G$ or $Q.H$ will be the chosen partitioning attributes.

As this example illustrates for relation Q, if a relation has two or more attributes with the same or close to the same weight, then the IR algorithm may not possibly make the best choice. In section 4.2, we extend our algorithm to consider this small set of attributes in a specialized selection and grouping algorithm.

More than one attribute could be chosen as the partitioning attribute set for a relation. In the example, attributes G and H could both be partitioning attributes for Q, especially if some sort of multi-dimensional indexing like grid file and its extensions [Ghand90, NHS84] are used. Multi attribute partitioning keys often result in far lower data distribution skews and should be considered if one attribute does not suffice. Multi attribute partitioning keys are also useful even in the absence of multi-dimensional indexing, when joins between two relations happen on multiple attributes, repeatedly. Thus hypothetically, if S and Q always joined with the clause $S.D = Q.GandS.E = Q.H$, then we could consider choosing $< G, H >$ as the partitioning key of Q. Our algorithm considers all such multi attribute partitioning keys.

4.1.3 Relation Group Algorithm

While a correct partitioning key of a relation with respect to a join helps to "direct" the tuples from the other relation to the correct nodes of this relation, true performance gain is achieved if the other relation is also partitioned compatibly for this join. Recall that in these cases, we also must ensure that the relations belong to the same group, i.e. share the same declustering strategy. In IR, since we have selected partition keys for each relation in the previous step, this phase of data placement groups the relations by finding a transitive closure of all joins performed on partitioning keys.

For example, let us use the query in figure 4 to represent the workload and the relationships between the relations. We use the partitioning attribute set of $\{Pkey\} = \{A, D, G \text{ or } H\}$. Relation R and S are joined on their partitioning attributes which indicates that they should be grouped together. Since a join with G or H is on the non-partitioning attributes of R and S, the relation Q can be assigned to a separate node group.

The only penalty with this approach is that one of the relations could be small, or for independent skew balancing purposes, we might want to decouple the two relations into separate groups which would normally have been grouped together. The first is not a problem, since small relations do not have a partitioning key. The second is also not likely to occur since attributes with high skews are eliminated by the filtering step as partition key candidates.

4.1.4 Declustering and Assignment

The declustering and assignment steps complete the data placement. We have based this step on ideas presented in past work [CABK88, MD94] with some extensions to the assignment phase. The main steps are:

- Break the relations into subpartitions whose number is larger than the total number of nodes as in Ghandeharizadeh's hybrid-range partitioning method [GDb90]. This excess in the number of subpartitions allows more flexibility in the assignment algorithm to balance data and loads across the partitions. The assignment algorithm will determine which subpartitions should be placed on which partitions. We use a simple assignment algorithm for the subpartition to partition mapping, e.g., round-robin assignment.

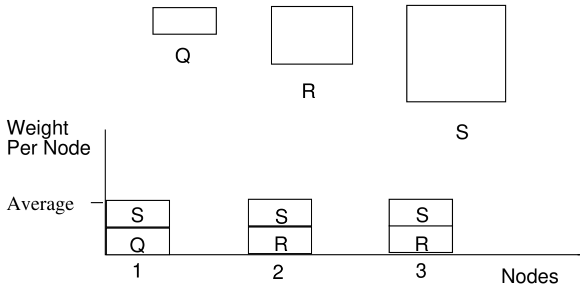

Figure 5: Example of the partition assignment algorithm on three nodes

- Large relations are placed on all nodes of the system [MD94]. Typically, the system is configured for large relations, so placing them on all nodes is required for capacity as well as performance reasons. Full declustering of the large relations was found to achieve the best performance for queries using these relations.
- For small relations, we use a variant of the Bubba assignment heuristic [CABK88]. The important variation in our method is that only the small relations are assigned in this step. The assignment details are described below.

We use the cardinalities of the small relations as their weight. This step effectively performs a Bin Packing [GJ78] of the relations so that the data is well balanced. Both the declustering and assignment of the small relations are computed together. The algorithm will search for a placement such that the weight at each node is close to an average weight. An average is calculated by summing the weights of all the small relations and dividing by the number of nodes. The algorithm sorts the relations from heaviest to lightest relation weight. It then tries to place the highest weighted relation. The degree of declustering of a relation will be the largest number of partitions which will allow the relation to be placed on the lightest weighted nodes keeping the total weight of each node below the average. The limit on a relation's degree of declustering is the number of nodes where each node could store one page of the relation.

Figure 5 offers a placement example. Relations Q, R, and S are to be placed on the nodes because they are of small cardinality. In this example, the weight of Q is one unit, the weight of R is two units, and the weight of S is three units. The average weight would be two units since there are three nodes in the system. The result of the placement is given in the lower portion of the figure. In this case, Q would have one partition, R would have two and S would have three.

4.2 Combined Relations Algorithm

Our independent relations algorithm, IR, works well if there is a clear choice for the partitioning attribute. In case this does not hold, picking at random from equal (or relatively equal) weighted attributes could result in a suboptimal behavior for a number of reasons.

The attribute weight selection does not consider the query optimizer. Let us say that IR chooses $R.A$ as the partitioning key.

- Consider a query in the workload $R.A = Q.D$ and $R.B = S.B$. The query optimizer could choose a plan where $R.B = S.B$ happens first, followed by $R.A = Q.D$, in which case the partitioning on $R.A$ does not help at all. Also, it is likely that $R.B$ could be a better partitioning candidate even if it has a lower aggregate weight than $R.a$.

- $R.A$ always joins with a relation which has very low cardinality, in which case the cost of the join is small and should be not be considered as equal to other joins during the attribute weighting step.

These points serve to illustrate the benefit of using the optimizer to better predict the partitioning key in cases of doubt.

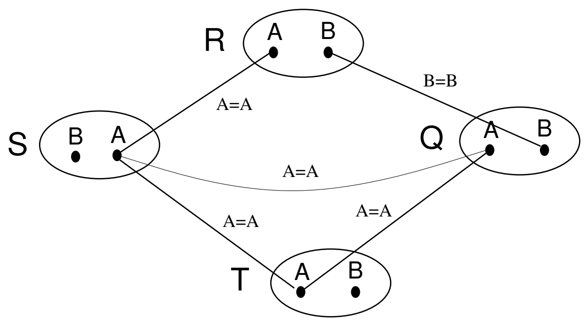

Figure 6: Example of a workload on a four relation DB (R, S, Q, T).

The IR algorithm also considers the relation grouping as a separate step. It is very much possible that because of independence in partitioning key selection, we could end up with more relation groups than required. Thus, considering partition key selection and relation grouping together could definitely lead to an advantage. Consider Figure 6 and assume that $R.B$ has a slightly higher weight than $R.A$. In that case, IR will choose $R.B$ as the partitioning key, resulting in two node groups – one containing S,T,Q and U, and the other containing R. Thus joins on $R.A$ and $R.B$ are not localized. However, we could have chosen $R.A$ as the partitioning key and obtained one relation group where only the join on $R.B$ is not localized. This decision is possible when both the partition key selection and relation grouping are combined together.

Figure 7 describes the main steps of the *Comb* algorithm. The *Comb* algorithm rectifies deficiencies of IR by (a) Exhaustively going through all partition keys which have the highest (or within a factor α of the highest weight) (b) determining the relation groupings for a particular choice of partitioning key set (c) using the query optimizer to generate the query plan and the estimated execution cost for each query based on this partitioning key and relation grouping and

```
Combined_Relations Algorithm
----------------------------
   /* calculate attribute weights as in the IR algorithm */
   1. Eliminate  all attributes of a relation not within
      a factor of Alpha from the highest weight candidate.

   2. Exhaustively consider all combinations of attributes
      that are not eliminated:
      i.   Find the relation grouping for the current candidate
           set.
      ii.  Call optimizer with candidate set and grouping to
           estimate execution times of queries.
      iii. Estimate Aggregate RT as the aggregate Response
           time of all queries.
      iv.  If this Aggregate RT is < best aggregate RT so far,
           {best partition key, relation grouping } is
           updated to the current one.

   3. Output the best partition key and relation grouping.

   /* Declustering and Assignment steps as in IR algorithm */
```

Figure 7: The Combined relations algorithm

(d) picking the particular partitioning key set that minimizes the overall estimated running time of the workload.

In a way, *Comb* is similar to the exhaustive algorithm we described in Section 3. The only difference is that the exhaustive algorithm goes through all possible partition keys, whereas *Comb* goes through only those that meet a certain threshold, determined by the attribute weight selection. From our experiments, we determined that a value of 0.5 for α effectively captures the important attributes of each relation. Typically, relations could have tens of attributes, whereas, only one or two qualify after the attribute weight assignments.

5 Experimental Analysis

We developed an analytical model of a Shared-Nothing parallel database system and used it to study the impact of the IR and *Comb* data placement algorithms. The analytical model consists of the following components: (i) system architecture, (ii) database schema and workload, (iii) query compiler, and (iv) a cost estimation component for the execution time and resource utilization of queries. Details of the components are presented in sections 5.1, 5.2, and 5.3. The model was used to study the overall impact of good partition key selection on workload performance as well as the

System Characteristic	Value
Number of nodes	64
Aggregate Network Bandwidth	10 MB/s to 100 MB/s
Processor speed	30 MIPS
Main memory per node	100 MB
Average I/O cost per page	16 ms
Number of pages prefetched	8
Page and message size	4 KB

Table 3: System Parameters

relative differences between the IR and *Comb* placement algorithms. The results of the experiments are described in Section 5.4.

5.1 System Parameters

Table 3 shows the important system parameters. We assumed a multicomputer system with 64 nodes and an interconnection network whose aggregate bandwidth was varied from 10 MB/s to 100 MB/s modeling a spectrum of topologies from bus-based to switch-based networks. We should mention that we translated network delay costs into CPU costs and added it to the Send/Receive CPU costs described in Table 4. Each node in the system is assumed to contain a CPU of 30 aggregate MIPS, 100 MB of main memory and adequate disks. The seek and rotational latency time for a new disk access is modeled as 16 ms and an average prefetch of 8 database pages is assumed.

5.2 Database and Workload Parameters

We considered using the current specification of TPC-D benchmark [Raab94] as the basis of our database and workload specifications. However, the workload description of TPC-D simplifies the partition key selection of most tables to their respective primary keys and the relative merit of the *Comb* algorithm cannot be quantified. Hence, we decided to generate artificial database and workload descriptions for the different experiments. The database was modeled after a simple Decision-Support type of database. In most of the experiments, the database contained *five* main relations; two of size 100 MB (small), two of size 10 GB (medium), and one of size 50 GB (large). All relations contained uniform row sizes of 108 bytes composed of five integer attributes, and a fixed character field.

To make the workload more interesting for our experiments, we generated random workloads

Operation	Processor Instructions
I/O per page	10000
Send/Receive protocol per page	15000
Send/Receive cost per byte	20 to 180
Join build phase per tuple	500
Join probe phase per tuple	1000
Scan and select cost per tuple	1000
Intermediate result (projection) per tuple	500

Table 4: CPU instruction counts for different operations

with queries having similar structures as the TPC-D queries, but with more than one join attribute per relation. This random generation allowed us to model a relatively complex workload where the configuration was not easily determined. We allowed at most three join attributes per relation. Queries were generated to join, using equi-joins, two to five randomly chosen relations. Each relation in a query was randomly involved in one to three equi-joins. In each of these joins, the other join relation was randomly chosen as was the two relations' join attributes. Each workload contained between eight and thirteen queries. Each random workload was considered a sample in our experiments. In our testing, enough samples were generated to obtain a 95% confidence interval for the performance metric of choice with an accuracy of 10%.

5.3 Workload Cost Analysis

Our model included a query compiler to translate the input queries into an execution plan. The execution plan consisted of *Select* operators at the leaves and intermediate *Join* operators. The Select phase had a fixed selectivity of 0.1 for each relation, while the Joins had varying selectivity parameters. Projections are performed at the earliest possible level of the execution plan. The joins are modeled as *Hash-based* join algorithms [SD90] consisting of a Build phase and a Probe phase. The relations involved in the join may require *Redistribution* phases prior to the actual join.

The cost of the Select and Join operators are estimated using the cost parameters described in Table 4. One important parameter is the send/receive instructions per byte of communication. This parameter counts not only the CPU cost incurred by the database system for send/receive but also includes the protocol costs and network latency costs. The network latencies have been transformed to appropriate CPU instruction counts. This parameter is varied from 20 instructions to 180 instructions to simulate different network and network protocol costs. While the figure of 180 instructions for the send/receive costs of a byte may look to be high, there are several reasons for considering such a figure. First, as mentioned above, we have bundled the network delay into this parameter. So, when we consider a relatively slower network (10 MB/s bandwidth),the send/receive cost increases. Second, the send/receive costs are not trivial in an industrial strength database system. The software must take care of different connections on behalf of different applications, provide for efficient packing of messages into and from buffers, and handle protocol error situations. All of this translates to a higher CPU cost for the send/receive parameter.

Cost formulas for the joins included calculations of whether one or both join relations are partitioned on the join attributes. If the relations are both partitioned on their join attributes and belong to the same relation group, the join cost is estimated as a local join. When one of the relations is partitioned on the join attribute, its nodes are selected as the destination nodes for redistributing the other relation. Priority is given to a larger relation's nodes as the destination of a redistributed relation for the ensuing join. Join result sizes were randomly chosen as a multiple of the size of the largest operand relation. With main memory constraints, relations not fitting into memory are assumed to always incur disk access. For hash-join estimates, if the build relation can fit into memory, I/O costs would be zero for the join since no overflow data needs to be temporarily placed on disk. We also assume that the data flows from one operation in the query to the next so that intermediate result storage would be minimized.

The performance metric used in the experiments was the *Weighted Average* of the *Execution Times* of the queries. This metric is most appropriate for the decision support databases being considered in this paper. In our experiments, all the main database tables are fully declustered across the 64 nodes of the system. The execution time of a query is computed as the largest of the sum of the I/O, CPU, and Network costs at any one node of the system and additional shared network overhead costs, if any. We ignore the cost of returning the results to the user since that cost is the same for any data placement technique. Therefore the estimated execution cost for query j, q_j, would be:

$$T(q_j) = \max_i (I/O_i + CPU_i + COMM_i) + Netovhd \tag{1}$$

where i is a node number, I/O_i is the total I/O cost, CPU_i is the total processor cost, $COMM_i$ is the total communication cost incurred for node i, and $Netovhd$ is the shared network overhead cost. The query-weighted Average Response Time (ART) of the workload is estimated by:

$$ART = \frac{\sum_{j=1}^{Q} f_j T(q_j)}{\sum_{j=1}^{Q} f_j} \tag{2}$$

where f_j is the frequency of execution for the query and there are Q queries in the workload.

5.4 Experimental Results

We performed two main experiments:

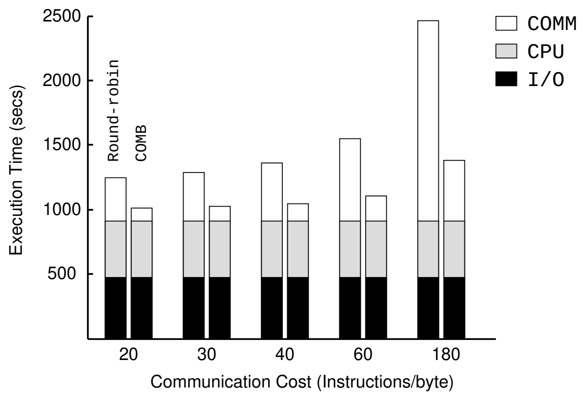

Figure 8: Performance Comparison of *Comb* placement with Round-Robin placement.

1. Performance based comparison of *useful* partition key selection and relation grouping against *round-robin* data placement. The round-robin scheme implies that all the base relations in a join query would require redistribution of their tuples prior to their first join. This experiment measures the performance improvement when using an intelligent data placement methodology.

2. Performance comparison between the IR and *Comb* algorithms. This experiment uses random and query graph biased workloads to quantify the performance differences between IR and *Comb*.

In all the figures, the X-axis represents different communication cost values based on the send/receive cost parameter. The y-axis represents the aggregate ART values from the different algorithms.

Figure 8 shows the performance difference between the *Comb* algorithm based placement and the round-robin placement technique. As described previously, the *Comb* algorithm would choose partition keys for the base relations so as to localize join operations between different relations. On the other hand, the round-robin scheme will require maximum communication of tuples of base relations. The difference in the ART varies from 30% for low send/receive costs to almost to almost 50% for higher communication costs. This experiment shows the importance of useful partition key selection and relation grouping (as performed by the *Comb* algorithm) for decision support database workloads.

Figure 9 shows the ARTs of randomly generated query workloads when they are placed using the IR and *Comb* algorithms. In the random workloads, any of the integer attributes of a relation has equal probability of being chosen as a join attribute. While there is a small difference between the ARTs for IR and *Comb*, the difference is of the order of a few percent (less than 5%). The reason for the relatively small improvement of *Comb* over IR is due to the random nature of the joins in the query workload. In such workloads, there is no special combination of partition keys that can

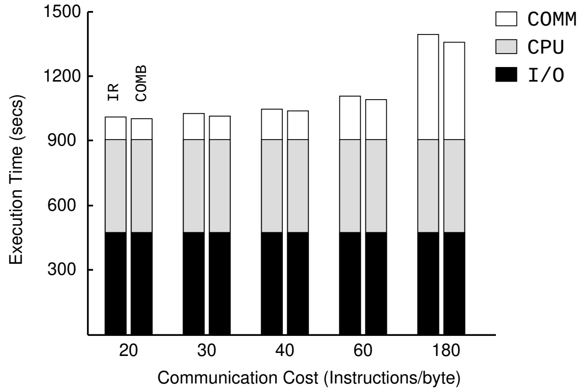

Figure 9: Performance Comparison of *Comb* and IR algorithms for random query workloads.

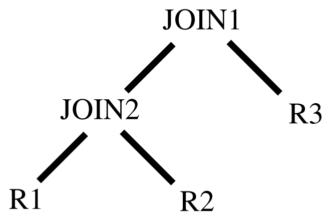

Figure 10: Query Execution Plan for join of R1, R2, R3 using clauses $R2.b = R3.b$ and $R1.a = R2.a$.

provide additional benefit. Thus, the IR algorithm, by choosing partition keys independently for each relation is as good as the *Comb* algorithm which tries to choose a good partition key group.

By examining the actual execution plans, the differences in the resource usages, and the overall query and workload performance differences, it was evident that the performance differences between IR and *Comb* were due to reduced communication in *Comb*. The reason for the communication difference is because of *Comb*'s ability to choose the partitioning attributes and relation groupings that fit with the query execution plans and especially with the ordering of the joins in the execution plans of a query.

An example of this phenomena is shown in Figure 10 where attributes a and b of relation $R2$ have the same weight. Assume we chose b as $R2$'s partitioning attribute. When we execute the lower level join, *JOIN2*, the fact that $R2$ was partitioned on b causes a redistribution of $R2$ on a so as to join with $R1$. The result of *JOIN2* then needs to be redistributed on b to join with $R3$ in *JOIN1*. However if $R2$ was partitioned on a we could reduce the redistribution needed for *JOIN2*, whereas the redistribution needed for *JOIN1* is always present.

We generated *biased* random workloads to quantify the performance improvement of the *Comb* algorithm over the IR algorithm. In the biased workload, we generated queries such that many of the lower level joins of the query execution plans would be based on a smaller set of attributes. The previous random workloads were changed so the join attributes at the lowest level of the query

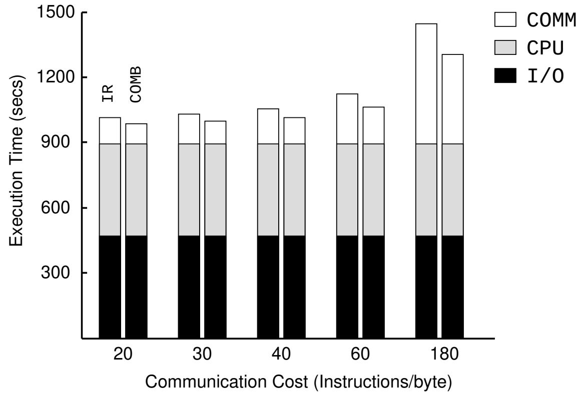

Figure 11: Performance Comparison of IR and *Comb* algorithms using biased random workloads.

plans would be dependent on an input probability distribution. A skew in this distribution would mean that only a few attributes are used as join attributes at the lowest level of the query plans.

Figure 11 shows the performance difference between the *Comb* and IR algorithms using the biased random workloads. As expected, the *Comb* algorithm improves the performance of the workload from 5% to 15% for the different communication costs. *Comb* uses the optimized execution plans during its partition key selection phase and is able to choose the better partition keys than IR due to this reason.

It should be noted that in some of the tested workloads a partitioning attribute chosen by *Comb* that reduced the lower level join costs was not the heaviest weighted attributes. The higher weighted attributes were involved in higher level joins, so IR tends to make a bad choice.

6 Conclusion

While several papers in the past have emphasized the importance of data placement in Shared-Nothing parallel database systems, most have focused on the declustering and assignment issues and paid passing attention to the very important task of selecting partitioning attributes meaningfully. This paper addresses the partition key selection problem and the associated issue of relation grouping for assignment in the context of an integrated data placement methodology.

Attributes which enable key optimizations in query processing (local joins, aggregations, distinct removal etc.) and hence reduce the execution costs of queries should be favored as partition keys. The *Independent Relations* (IR) algorithm uses the query graph and relative weights for different types of parallel query optimizations to generate relative importance of the attributes of each relation in the database. Each relation is considered in isolation, and the highest weighted attribute (or combination) is recommended as the partitioning key of the relation. This algorithm has a polynomial time complexity both in terms of the number of queries and the number of relations in the database. After the partition key selection phase, the IR algorithm will also perform relation grouping based on selected partitioning attributes and assignment of relations.

While the IR algorithm is a very good placement algorithm for simple database workloads, it is less efficient in complex query workloads when there may be several partition key candidates for each relation. We described the *Combined Relations* (*Comb*) algorithm which uses the IR's partition key selection phase to reduce the number of partition key candidates for each relation and then uses a combinatorial algorithm to consider combinations of keys across different tables. The *Comb* algorithm uses the query optimizer as the basis for choosing partition key combinations. Therefore, it is able to choose a very useful combination of partition keys that reduces overall workload cost for complex workloads. The *Comb* algorithm is also low-cost since it only considers a few candidates (the output of the IR step) for each relation in its combinational step. The *Comb* algorithm iterates between the partitioning attribute and relation grouping stages to provide an even better placement selection.

We presented experimental results based on an analytical model to show the importance of partition key selection and relation grouping steps of data placement. We showed that the *Comb* based placement can produce 30 to 50% performance improvement over simple round-robin placement. We also studied the relative differences between the IR and *Comb* algorithms using random and biased workloads. *Comb* was shown to perform better when the queries were biased to favor certain attributes of relations with higher join costs. The IR algorithm cannot make such a determination since it does not use query optimization in its algorithm.

Future Work

We can make both our algorithms as well as the model for evaluation more sophisticated. Some of the possible extensions are:

- Determining the response time taking into consideration the actual data skew (note that our optimizer assumes a uniform data distribution)
- Extending IR to modify the attribute weights based on their expected *importance during execution*, e.g., if t1.a = t2.b is executed repeatedly, but t2.b is small, then do not give a high weight to t1.a. *Selectivity* inputs are also critical at this stage.
- Making the algorithms more robust by detecting bad queries or repeated weight assignments that provide no value add. For example, an adversary could break our data placement tool by adding clauses which do not change the query result.

Another useful investigation is the use of the data placement algorithm to reconfigure the system during run-time. This problem is even more interesting if we consider that the reconfiguration, or reorganization of the data, must be executed concurrently with the normal database workload. A reorganization should thus be undertaken only when the performance benefits after reconfiguring the database outweigh the intrusion costs that the reorganization places on the system. The interesting issue is the following: suppose we periodically run the data placement algorithms on the workloads as they change, and supposing that the output of the algorithm indicates a data organization different from the existing one. Then, is the cost of reorganization justified? In other words, not only should the tool give its "best" output, it should also be able to predict the performance of a given "input". For this, the analytical model of the tool would need considerable enhancement.

References

[BZ93] Chaitanya Baru and Daniel C. Zilio, "Data Reorganization in Parallel Database Systems", *IEEE Workshop on Advances in Parallel and Distributed Systems*, pp. 102-107, IEEE Computer Society, Princeton, NJ, Oct. 1993.

[B+90] H. Boral *et al*, "Prototyping Bubba, a highly parallel database system", *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, pp. 4-24, Mar. 1990.

[CABK88] George Copeland, William Alexander, Ellen Boughter, and Tom Keller, "Data Placement in Bubba", *Proc. of the ACM SIGMOD Conf.*, pp. 99-108, ACM Press, Chicago, June 1988.

[CC93] L. Chambers and D. Cracknell, "Parallel Features of NonStop SQL", *Parallel and Distributed Information Systems Conf.*, pp. 69-70 IEEE Computer Society, San Diego, Jan. 1993.

[Clay93] D. Clay, "Informix Parallel Data Query" *Parallel and Distributed Information Systems Conf.*, pp. 71-72, IEEE Computer Society, San Diego, Jan. 1993.

[DBPE94] Database Technology Group, "Database 2 AIX/6000 Parallel Technology", *Technical brochure*, IBM SWSD, North York, Canada, Mar. 1994.

[DG85] David J. DeWitt and Robert Gerber, "Multiprocessor Hash-Based Join Algorithms", *Proc. of the Int. Conf. on VLDB*, pp. 151-164, Morgan-Kaufmann, Stockholm, Aug. 1985.

[DG92] David DeWitt and Jim Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Comm. of the ACM*, pp. 85-98, ACM Press, June 1992.

[D+90] David J. DeWitt em et al, "The GAMMA database machine project", *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, pp. 44-62, Mar. 1990.

[DGGHKM86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine", *Proc. of the Int. Conf. on VLDB*, pp. 228-237, Morgan-Kaufmann, Kyoto, Aug. 1986.

[FK92] Felipe Carino Jr. and Pekka Kostamaa, "Exegesis of DBC/1012 and P-90 - Industrial Supercomputer Database Machines", *PARLE'92 Conf. Parallel Architectures and Languages Europe*, Springer-Verlag, Paris, June 1992.

[FM89] Christos Faloutsos and Dimitrios Metaxas, "Declustering Using Error Correcting Codes", *Proc. of the ACM Symp. on Principles of Database Systems*, pp. 253-258, ACM Press, Philadelphia, March 1989.

[GDa90] Shahram Ghandeharizadeh and David DeWitt, "A Multiuser Performance Analysis of Alternative Declustering Strategies", *IEEE Int. Conf. on Data Engineering*, pp. 466-475, IEEE Computer Society, Los Angeles, Feb. 1990.

[GDb90] Shahram Ghandeharizadeh and David DeWitt, "Hybrid-Range Partitioning: A New Declustering Strategy for Multiprocessor Database Machines", *Proc. of the Int. Conf. on VLDB*, pp. 481-492, Morgan-Kaufmann, Brisbane, Australia, Aug. 1990

[Ghand90] Shahram Ghandeharizadeh, "Physical Database Design in Multiprocessor Database Systems", *Tech. Report #964 (PhD Thesis)*, Computer Sciences Dept., Univ. of Wisconsin-Madison, Sept. 1990.

[Ghand91] Shahram Ghandeharizadeh, Luis Ramos, Zubair Asad, and Waheed Qureshi, "Object Placement in Parallel Hypermedia Systems", *Proc. of the Int. Conf. on VLDB*, pp. 243-254, Morgan-Kaufmann, Barcelona, Sept. 1991.

[GJ78] M.R. Garey and D.S. Johnson, "Computers and Intractability: A guide to the theory of NP-Completeness", W.H. Freeman and Co., 1978.

[Hsaio90] Hui-I Hsaio, "Performance and Availability in Database Machines with Replicated Data", *Tech. Report #963*, Computer Sciences Dept., University of Wisconsin-Madison, Aug. 1990.

[HL90] Kien A. Hua and Chiang Lee, "An Adaptive Data Placement Scheme for Parallel Database Computer Systems", *Proc. of the Int. Conf. on VLDB*, pp. 493-506, Morgan-Kaufmann, Brisbane, Australia, Aug. 1990.

[KP88] Myoung Ho Kim and Sakiti Pramanik, "Optimal File Distribution for Partial Match Retrieval", *Proc. of the ACM SIGMOD Conf.*, pp. 173-182, ACM Press, Chicago, June 1988.

[KS94] Ralph Kimball and Kevin Strehlo, "Why Decision Support FAILS and How to FIX it", *Datamation*, pp. 1994, June 1994.

[Linder93] B. Linder, "Oracle Parallel RDBMS on Massively Parallel Systems", *Parallel and Distributed Information Systems Conf.*, pp. 67-68, IEEE Computer Society, San Diego, Jan. 1993.

[LS92] Jianzhong Li and Jaideep Srivastava, "CMD: A Multidimensional Declustering Method for Parallel Database Systems", *Proc. of the Int. Conf. on VLDB*, Morgan-Kaufmann, Vancouver, Aug. 1992.

[LKB87] Miron Livny, Setrag Khoshafian, and Haran Boral, "Multi-Disk Management Algorithms", *Proc. of the ACM SIGMETRICS Conf.*, pp. 69-77, ACM Press, May 1987.

[MD94] Manish Mehta and David J. DeWitt, "Data Placement in Shared-Nothing Parallel Database Systems", *Proc. of the ACM SIGMOD Conf.*, pp. ???, ACM Press, Minneapolis, May 1994.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable Symmetric Multikey File Structure", *ACM Trans. on Database Systems*, pp. 38-71, ACM Press, March 1984.

[Padman92] Sriram Padmanabhan, "Data Placement in Shared-Nothing Parallel Database Systems", *PhD Thesis*, EECS Dept., Univ. of Michigan-Ann Arbor, 1992.

[PB92] Sriram Padmanabhan and Chaitanya Baru, "Data Placement in Shared-Nothing Parallel Database Systems", *Conf. on Information and Knowledge Management*, pp. 345-352, Baltimore, 1992.

[Perna94] Janet Perna (editor), *DB2 AIX/6000 Parallel Technology (Customer Information)*, IBM Corporation, 1994.

[Raab94] Francois Raab (editor), *TPC Benchmark D (Decision Support) Working Draft 6.5*, Transaction Processing Council (TPC), Feb. 1994.

[SD90] D. Schneider and D. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines", *Proceedings of the 16th Intl. VLDB Conf.*, Brisbane, Australia, 1990.

[SG84] Kenneth Salem and Hector Garcia-Molina, "Disk Striping", *Tech. Report EECS-TR-332-84*, Dept. of Computer Science, Princeton University, Dec. 1984.

[TPCA91] Transaction Processing Council, *TPC Benchmark A Standard Specification*, Sept. 1991.

[WZS91] Gerhard Weikum, Peter Zabback, and Peter Scheuermann, "Dynamic File Allocation in Disk Arrays", *Proc. of the ACM SIGMOD Conf.*, pp. 406-415, ACM Press, Denver, May 1991.