Avoiding Sorting and Grouping In Processing Queries*

Xiaoyu Wang Mitch Cherniack

Department of Computer Science
Brandeis University
Waltham, MA, USA, 02450
{wangxy, mfc}@cs.brandeis.edu

Abstract

Sorting and grouping are amongst the most costly operations performed during query evaluation. System R [6] used simple *inference* strategies to determine orderings held of intermediate relations to avoid unnecessary sorting, and to influence join plan selection. Since then, others have proposed using integrity constraint information to infer orderings of intermediate query results. However, these proposals do not consider how to avoid grouping operations by inferring *groupings*, nor do they consider *secondary orderings* (where records in the same group satisfy some ordering). In this paper, we introduce a formalism for expressing and reasoning about *order properties*: ordering and grouping constraints that hold of physical representations of relations. In so doing, we can reason about how the relation is ordered or grouped, both in terms of primary and secondary orders. After formally defining order properties, we introduce a *plan refinement* algorithm that infers order properties for intermediate and final query results on the basis of those known to hold of query inputs, and then exploits these inferences to avoid unnecessary sorting and grouping. We then show empirical results demonstrating the benefits of plan refinement, and show that the overhead that our algorithm adds to query optimization is low.

1 Introduction

Sorting and grouping are amongst the most costly operations performed during query evaluation. As blocking operations, they also hinder pipelining. System R [6] was the first system to recognize the benefit of detecting orderings that held of base relations and intermediate query results to avoid unnecessary sorting and reduce the cost of join processing (exploiting so-called "interesting orders"). Others have since proposed the use of integrity constraints to *infer* orderings (e.g., [9], [3], [8]). For example, in their seminal paper on *order optimization*, Simmen et al [8] show how functional dependencies and selection predicates can be used to determine how orderings get propagated from inputs to outputs of joins.

In this paper, we present a formal and comprehensive approach to order optimization. While in the same spirit as [8], we make the following novel contributions:

1. Aside from orderings, we also infer how relations are *grouped* (i.e., how records in relations are clustered according to like value of certain attributes). Just as inference of orderings can make it possible to avoid sorting, inference of groupings can make it possible to avoid hash-based grouping algorithms (e.g., prior to duplicate elimination or group-wise aggregate computations). We also consider *secondary* orderings and groupings (i.e., those which hold within each group determined by a primary ordering or grouping). By inferring secondary orderings and groupings, it is possible to avoid unnecessary sorting or grouping over multiple attributes. Also, in some cases one can use secondary orderings known of an operator's input to infer primary orderings of of its output.

2. We present a formal semantics for *order properties* (i.e., primary and secondary orderings and groupings), thereby making it easier to reason about and verify inference techniques (Section 3).

3. We introduce a *plan refinement* algorithm that decorates query plans produced by Postgres [10] with inferred orderings and groupings and then refines these plans by removing unnecessary sorting and grouping operations (Section 4).

4. We empirically show the benefits of plan refinement and the low overhead it adds to the cost of

*This work has been supported by the NSF under the grant IIS-9984960.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

```
SELECT c_custkey, COUNT (*)
FROM Customer, Supplier
WHERE c_nationkey = s_nationkey
GROUPBY c_custkey
```

Figure 1: A Simple Example Query

query optimization (Section 5).

We begin in Section 2 by motivating our approach to order optimization by working through the optimization of a simple example query based on the TPC-H schema using the grouping and secondary ordering inference techniques presented here.

2 Motivation

We first demonstrate the benefits of inferring *grouping* and *secondary orderings* with a simple example. The SQL query shown in Figure 1, which uses the manufacturer-based schema of the TPC-H benchmark, associates every customer in the `Customer` table with the number of suppliers from the `Supplier` table that are situated in the same country. Such a query might be used to determine how many suppliers could supply each customer directly without having to go through customs. Figure 5 shows an execution plan for this query as generated by Postgres [10] (5a), and the same Postgres plan after modification according to our inference techniques (5c). (Figure 5b can be ignored for the time being.) Note that both plans execute a sort-merge join on `Customer` and `Supplier` and aggregate (`Group`) on the result. However, prior to aggregation the Postgres plan of Figure 5a first sorts the join result on the grouping attribute, `c_custkey` so as to be able to aggregate over groups in a single pass. **But one-pass aggregation requires data only to be grouped and not sorted!** Our approach allows us to infer that the result of the sort-merge join will always be grouped on `c_custkey`, making the sorting step (or a hash-based grouping step, as might be used in other systems [1]) unnecessary. When applied to the query plan of Figure 5a, our refinement algorithm would detect the unneeded `Sort` operation and produce the plan of Figure 5c.

In Section 4.2, we give a detailed explanation of how we are able to infer that the result of the sort-merge join is guaranteed to be grouped on `c_custkey`. Summarized briefly, this result follows from the following reasoning:

1. The sort-merge equijoin produces a result that is sorted (and hence grouped) on its join attributes (`c_nationkey`). Further, sort-merge also ensures that output tuples that are in the same group with respect to `c_nationkey` are themselves grouped on the key of the outer relation (`c_custkey`), as output tuples with the same value of `c_custkey` are produced all at once as a result of processing a single tuple from the outer relation. We express this "grouping within groups" as "$\texttt{c_nationkey}^G \rightarrow \texttt{c_custkey}^G$".
2. Because `c_custkey` functionally determines `c_nationkey`, it must be the case that the result of the join is also grouped on `c_custkey`. Therefore, it is not necessary to sort or group this result prior to aggregation, as the necessary grouping is already satisfied.

Note that the above optimization required reasoning about both groupings and secondary orderings. What allowed us to determine that preprocessing was unnecessary prior to aggregation was the inference that a join result was *grouped*. But we would not have been able to make this inference had we not recognized the relevant grouping (on `c_custkey`) as a secondary grouping following the primary ordering (and hence grouping) on `c_nationkey`. The impact of this optimization is significant – when applied to a database instance of 150,000 rows in `Customer` and 10,000 rows in `Supplier`, the plan shown in Figure 5b outperforms that shown in Figure 5a by an order of magnitude (as we will show in Section 5).

In general, the inference of orderings or groupings of intermediate query results can be exploited in the ways listed below. Note that an ordering on some attribute A is a special case of a grouping on the attribute A. Therefore, all the applications below involving inferred groupings also benefit from inferred orderings.

1. Inferred orderings can be used to avoid sorting as a preprocessing step to sort-merge join.
2. Inferred orderings can be used to avoid sorting when processing `ORDER BY` clauses of SQL queries. Inferred secondary orderings can be used to avoid sorting when processing `ORDER BY` clauses that list multiple attributes.
3. Knowledge of inferred orderings can be exploited in making decisions about how to "push down sorts" past joins. This enables sorting to be performed on smaller input relations rather than larger relations. Knowledge of inferred groupings similarly enables "pushing down group-by operations" as was discussed in [12] and [2].
4. Inferred groupings can be used to avoid sorting or hashing prior to computing aggregates for `GROUP BY` clauses.
5. Inferred groupings can be used to reduce the cost of projection with duplicate elimination, provided that projection is over the grouped attribute. In this case, projection and duplicate elimination can be completed in a single pass. Similarly, any other operation requiring duplicate elimination needs no sorting or hash-based grouping preprocessing step.

6. Inferred groupings can be used to reduce the cost of evaluating selection queries of the form, $\sigma_{A=k}(R)$ in the absence of indexes or an ordering on A. Provided that R is grouped on A, a sequential scan can be terminated prior to completion, once the first tuple is seen whose value for A does not equal k, and which follows a tuple whose value for A does equal k. This is a generalization of the standard technique for searching for a value of a key attribute on an unordered, unindexed file, and similarly would reduce the cost of searching on average by a factor close to 2.

7. Inferred secondary orderings or groupings can be used to infer new primary orderings or groupings. The optimization of the query of Figure 1 illustrated this. We will see other examples in Section 4.

3 Order Properties

We begin in Section 3.1 by briefly reviewing some helpful identities from [8] concerning functional dependencies and keys. A novel formalism for order properties is presented in Section 3.2, and some example order property inference rules are shown in Section 3.3.

3.1 Functional Dependencies and Keys

We use the notation, $FD_{X \rightarrow B}(R)$ to say that the functional dependency, $X \rightarrow B$ holds of relation R (for set of attributes, X and attribute B), and $FD_{A \rightarrow B}(R)$ as shorthand for $FD_{\{A\} \rightarrow B}(R)$. (We reserve variable names $A, B, C, \ldots$ to name individual attributes and $X, Y, \ldots$ to name sets of attributes.) Similarly, $Key_X(R)$ says that attribute set X is a key of relation R, and $Key_A(R)$ is shorthand for $Key_{\{A\}}(R)$.

It has long been known that functional dependencies (unless they involve projected attributes) get propagated through joins [4], [8]. We express this with the following axiom.

Axiom 3.1 (FD Propagation) *For relations, R and S, sets of attributes, $X, Y \subseteq R \cup S$[1], and join predicates, p:*

$$(FD_{X \rightarrow Y}(R) \vee FD_{X \rightarrow Y}(S)) \;\Rightarrow\; FD_{X \rightarrow Y}(R \bowtie_p S).$$

As was discussed in [8], when the join predicate p is an equijoin whose join attributes are contained in X or Y, functional dependencies that involve attributes from both R and S can be inferred. For example, the following is a useful identity derivable from Axiom 3.1.

Identity 3.1 *For all relations, R and S and attributes $A \in R$ and $B, C \in S$:*

$$FD_{B \rightarrow C}(S) \Rightarrow FD_{B \rightarrow A}(R \bowtie_{A=C} S).$$

[1]We assume wlog that $R \cap S = \emptyset$.

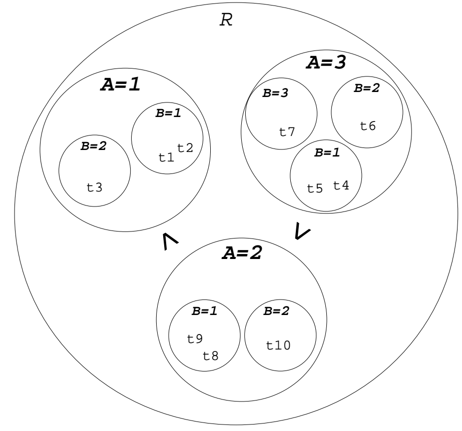


Figure 2: An Illustration of $A^O \rightarrow B^G$

Key propagation through joins is a standard topic in introductory database courses [7]. We capture this via the axiom,

Axiom 3.2 (Key Propagation) *For relations, R and S, sets of attributes $X \subseteq R$ and $Y \subseteq S$, and join predicate p:*

$$(Key_X(R) \wedge Key_Y(S)) \;\Rightarrow\; Key_{X \cup Y}(R \bowtie_p S).$$

As with functional dependencies, key attributes can be reduced in the case of equijoins involving join attributes which are also key attributes, as expressed in the identity below:

Identity 3.2 *For relations, R and S, set of attributes $X \subseteq R$, and attributes $A \in X$ and $C \in S$:*

$$Key_X(R) \Rightarrow Key_{(X \cup Y)-\{A\}}(R \bowtie_{A=C} S).$$

3.2 Order Properties

Informally, order properties have the form,

$$A_1^{\alpha_1} \rightarrow A_2^{\alpha_2} \rightarrow \ldots \rightarrow A_n^{\alpha_n}$$

such that each A_i is an attribute, each α_i either specifies an ordering ($\alpha_i = O$)[2] or a grouping ($\alpha_i = G$), and $A_1^{\alpha_1}$ is a primary ordering or grouping, $A_2^{\alpha_2}$ is a secondary ordering or grouping and so on. A relation whose physical representation satisfies such an order property can be viewed as a nested set of (potentially ordered) equivalence classes, as the following example illustrates.

Suppose that $R = (A, B)$ consists of 10 tuples, $t_1, \ldots, t_{10}$, and that its physical representation satisfies the order property, $A^O \rightarrow B^G$: an ordering on A

[2]To simplify exposition, we assume all orderings for the purposes of this paper to be ascending. Inference techniques for descending orderings are the obvious analogs to those we present here.

followed by a grouping on B. This is illustrated in Figure 2. Circles in this figure denote groups (the outermost circle denoting the group consisting of all tuples in R), and orderings between groups are shown with '<'. Thus, tuples $t_4, \ldots, t_7$ comprise a group because they have a common value for A ($A = 3$), and t_4 and t_5 comprise a group within that group because they also share the same value for B ($B = 1$).

The primary ordering (A^O) says that the group of tuples with $A = 1$ precedes the group of tuples with $A = 2$, which in turn precedes the group of tuples with $A = 3$. From this we can infer, for example, that both t_1 and t_3 precede t_9, but we can infer no ordering between t_1 and t_3. The secondary ordering (B^G) says that within each group of tuples with like values of A, tuples are clustered together if they have the same value for B. Thus, t_1 can precede t_2 or t_2 can precede t_1, but they must be adjacent. Thus, there are many permutations of R that satisfy this order property (and most order properties in general). Two example permutations that satisfy this order property are shown below.

- $t_3, t_1, t_2, t_{10}, t_8, t_9, t_6, t_7, t_4, t_5$
- $t_1, t_2, t_3, t_9, t_8, t_{10}, t_4, t_5, t_6, t_7$.

Note that the latter permutation also satisfies the ordering, $A^O \rightarrow B^O$.

More formally, we say that a *physical representation of a relation* (or just *physical relation*) is a list of tuples ordered according to their placement as records in a file.[3] For any physical relation, R, this ordering is expressed with the total order, "$\rhd_R$" Thus, we have the following definition for a physical relation.

Definition 3.1 (Physical Relations) *A physical relation, R, is a list of tuples*

$$t_1 \rhd_R \ldots \rhd_R t_n$$

such that "$t_i \rhd_R t_j$" holds of records t_i and t_j if t_i immediately precedes t_j in the file representation of R.

Thus, we begin from the assumption that each relation has an associated dedicated file, and that records are stored row-wise.

For convenience, we denote the irreflexive, asymmetric transitive closure of "$\rhd_R$" as "$\rhd_R^+$". Thus, $t_i \rhd_R^+ t_j$ if t_i precedes t_j (not necessarily immediately) in the file representation of R.

Order properties are formulated with an algebra of constructors whose signatures are shown in Figure 3. An order property (*Ord*) is defined recursively as either an *empty* order property $\perp$, or the combination ("$\rightarrow$") of a basic order property (*BOrd*) with an order property. Basic order properties are either orderings (A^O)

[3]Note that a file may not actually be stored, as in the case of pipelined results from intermediate queries.

Expn	Signature		
$\perp$		$\rightarrow$	*Ord*
$__ \rightarrow __$	*BOrd*, *Ord*	$\rightarrow$	*Ord*
$__^O$	*Att*	$\rightarrow$	*BOrd*
$__^G$	*Att*	$\rightarrow$	*BOrd*

Figure 3: Signatures for Ordering Constructs

or groupings (A^G) on individual attributes. Thus in general, order properties have the form,

$$(A_1^{\alpha_1} \rightarrow (A_2^{\alpha_2} \rightarrow (\ldots \rightarrow (A_n^{\alpha_n} \rightarrow \perp))))$$

for some set of simple order properties, $A_i^{\alpha_i}$. For convenience, we express this using the shorthand,

$$A_1^{\alpha_1} \rightarrow A_2^{\alpha_2} \rightarrow \ldots \rightarrow A_n^{\alpha_n}.$$

Also, given

$$\begin{aligned} o &= A_1^{\alpha_1} \rightarrow A_2^{\alpha_2} \rightarrow \ldots \rightarrow A_k^{\alpha_k}, \text{ and} \\ o' &= A_{k+1}^{\alpha_{k+1}} \rightarrow A_{k+2}^{\alpha_{k+2}} \rightarrow \ldots \rightarrow A_n^{\alpha_n} \end{aligned}$$

we use the shorthand "$o \rightarrow o'$" to express the order property,

$$A_1^{\alpha_1} \rightarrow A_2^{\alpha_2} \rightarrow \ldots \rightarrow A_n^{\alpha_n}.$$

We sometimes refer to "$o_1 \rightarrow o_2$" as the *concatenation* of order properties o_1 and o_2. Finally, for any *set* of attributes, $X = \{A_1, \ldots, A_n\}$, we use the notation, X^G as shorthand for

$$A_1^G \rightarrow A_2^G \rightarrow \ldots \rightarrow A_n^G.$$

The formal semantics of each order property constructor is defined in terms of the relationship, "$O_o(R)$" which holds if physical relation R satisfies order property o. Axioms defining order properties in terms of this relation are shown below. Axiom 3.3 says that every physical relation satisfies the empty order property. Axiom 3.4 says that R is ordered on attribute A ($O_{A^O}(R)$) if tuples with lower values for A appear earlier in the file than tuples with higher values for A.

Axiom 3.3 (Empty Order) *For all physical relations, R,*

$$O_\perp(R) \Leftrightarrow \textit{TRUE}.$$

Axiom 3.4 (Basic Ordering) *For all physical relations R and attribute $A \in R$,*

$$O_{A^O}(R) \Leftrightarrow \forall\ t, u \in R\ (t.A < u.A \Rightarrow t \rhd_R^+ u).$$

Axiom 3.5 says that R is grouped on attribute A ($O_{A^G}(R)$) if all tuples with the same value for A are clustered together. This is captured formally by saying that for any two tuples with the same value for A, a, all tuples that lie between them also have a value for A of a.

Axiom 3.5 (Basic Grouping) *For all physical relations, R and attribute $A \in R$,*

$$O_{A^G}(R) \Leftrightarrow \forall\ t,u \in R\ (t.A = u.A \Rightarrow p\ (t,u,R))$$

such that

$$p\ (t,u,R) \Leftrightarrow \forall\ v \in R\ ((t \rhd^+_R v \rhd^+_R u) \Rightarrow v.A = t.A).$$

Axiom 3.6 defines secondary order properties, saying that R satisfies $A^\alpha \rightarrow o$ iff:

- R satisfies the basic order property, A^α, and
- every group of R with a common value for A satisfies order property, o.

Axiom 3.6 (Secondary Order Properties) *For any physical relation, $R = t_1 \rhd_R \ldots \rhd_R t_n$, and attribute $A \in R$, let $R_{[A=c]}$ be the group of R tuples whose value for $A = c$, in the order that they are found in R. More precisely,*

$$R_{[A=c]} \quad = \quad t_{j_1} \rhd_{R_{[A=c]}} \ldots \rhd_{R_{[A=c]}} t_{j_m}$$

such that:

- $\{t_{j_1}, \ldots, t_{j_m}\} = \{t_i | 1 \leqslant i \leqslant n, t_i\ .A = c\}$*, and*
- $(t_{i_j} \rhd_{R_{[A=c]}} t_{i_k}) \Rightarrow (t_{i_j} \rhd_R t_{i_k})$.

Then, for any order property, o,

$$O_{A^\alpha \rightarrow o}(R) \Leftrightarrow O_{A^\alpha}(R)\ \wedge\ \forall\ c\ (O_o(R_{[A=c]})).$$

Figure 4 shows some useful identities derivable from the ordering property axioms above. Identity #1 says that for any order property that holds of a physical relation, all prefixes of that order property also hold of R. In most cases, it is not useful to drop suffixes of order properties as ordering information gets lost. However, in some cases these suffixes are trivial and therefore yield no interesting ordering information. For example, if A is a key for R, then any order property over R of the form, $A^G \rightarrow o$ is trivial as all groups denoted by A^G consist of a single tuple, and hence secondary orderings over these groups are meaningless. (Similarly, $A^O \rightarrow o$ can be reduced to A^O without losing non-trivial information.) Therefore, we always assume that order properties are normalized such that no order property includes a key unless it is ordered, in which case it appears at the end. Identity #2 of Figure 4 says that an ordering on any attribute implies a grouping on that attribute. Identity #3 says that if X functionally determines B, and an order property that includes all attributes in X (ordered or grouped) appearing before B^α, then B^α is superfluous. Identity #4 is a special case of identity #3, covering the case where X consists of a single attribute. Identity #5 says that the grouping of an attribute that is functionally determined by the attribute that follows it in the order property is superfluous. Correctness proofs of the identities of Figure 4 can be found in [11].

1. $O_{o \rightarrow o'}(R) \Rightarrow O_o(R)$
2. $O_{o \rightarrow B^O \rightarrow o'}(R) \Rightarrow O_{o \rightarrow B^G \rightarrow o'}(R)$
3. $FD_{X \rightarrow B}(R), O_{o \rightarrow B^\alpha \rightarrow o'}(R), X \subseteq \mathit{atts\ in\ o} \Rightarrow O_{o \rightarrow o'}(R)$
4. $FD_{A \rightarrow B}(R), O_{o \rightarrow A^\alpha \rightarrow o' \rightarrow B^\beta \rightarrow o''}(R) \Rightarrow O_{o \rightarrow A^\alpha \rightarrow o' \rightarrow o''}(R)$
5. $FD_{A \rightarrow B}(R), O_{o \rightarrow B^G \rightarrow A^G \rightarrow o'}(R) \Rightarrow O_{o \rightarrow A^G \rightarrow o'}(R)$

Figure 4: Some Identities Based on Axioms 3.3 – 3.6

3.3 Order Property Inference

Table 1 shows a set of inference rules for determining what order properties hold as a result of executing 4 different join algorithms: nested loop join, sort-merge join, simple-hash-join and order-preserving hash join [5]. For all rules expressed in this table, R and S are physical relations with attributes, A and B, and sets of attributes, X and Y respectively, o and o' are order properties, and p is a join predicate. Again, correctness proofs of these rules, based on axioms 3.2–3.5 and the operational semantics of the join algorithms can be found in [11].

Rule 1 of Table 1 (observed previously in [8] and [5]) says that the output relation from a nested loop join inherits the ordering of the outer (input) relation. Rule 1 also holds of a *simple hash join*, as is implemented in Postgres [10] (Rule 2). Rule 3 is a specialized form of Rule 1 that holds when the outer relation contains a key, X. In this case, each group of tuples in the output with common values for X will also be ordered on the inner relation's ordering.

Like nested-loop join, the output of merge join is always ordered by the order property of the *outer relation*. And like nested-loop join, if the outer relation contains a key, then the output relation is ordered by the concatenation of the order properties of the outer and inner relations, separated by a grouping on the outer relation key. This is expressed formally in Rule 4. Note that this rule is more specific than the corresponding rule for nested loop join (Rule 3) because input relations to merge joins are always sorted on their join attributes and because we are assuming merge joins are equijoins only.

Order-preserving hash joins were introduced by Claussen et al. [5] as a variant of hash join that supports early sorting. The algorithm is based on Grace hash join and exploits a prior sorting of the outer (probe) relation to produce a result that is similarly ordered. This variant of hash join therefore resembles nested loop and sort-merge join in preserving orderings of outer relations. It is important to note that order-preserving hash join does preserve orderings, but does not preserve groupings held of the outer relation. That

is, if the order property satisfied by the outer relation is of the form,

$$A_1^O \rightarrow \ldots \rightarrow A_{k-1}^O \rightarrow A_k^G \rightarrow o$$

such that A_k is the leftmost grouped attribute, then only $A_1^O \rightarrow \ldots \rightarrow A_{k-1}^O$ is preserved in the outer relation: all order properties following and including the first basic grouping property (A_k^G) are lost. This is expressed formally in Rule 5.

4 Order Property Optimization

We have designed and implemented a *plan refinement* algorithm that refines query plans produced by Postgres [10] so as to eliminate unnecessary sorting and grouping operations.[4] The plan refinement algorithm uses known functional dependencies, key properties and order properties of base relations to infer order properties of intermediate query results. It then uses inferred order properties to determine which unnecessary sort operations can be removed. We first summarize relevant Postgres plan operators in Section 4.1, and then describe the algorithm and present examples of refined query plans in Section 4.2.

4.1 Postgres Plan Operators Summarized

Table 2 presents the plan operators (nodes) of Postgres that are relevant to our plan refinement algorithm. (A plan operator is irrelevant to our algorithm if it generates no new functional dependency, key or order property information from child or parent nodes. Operators that are irrelevant to our plan refinement algorithm include: `Material`, `Subplan`, `SubqueryScan`, `TIDScan`, `Limit` and `Result`).[5]

The data structures for all plan nodes in Postgres include the following fields:

- $\texttt{inp}_1, \ldots \texttt{inp}_n$: the fields contained in all input tuples to the node,
- `left`: the left subtree of the node (set to `Null` for leaf nodes and `Append`),
- `right`: the right subtree of the node (set to `Null` for leaf nodes, unary operators and `Append`).

As well, additional operator-specific fields provided by Postgres and used by our refinement algorithm are listed in the *Attributes* column of Table 2 (modulo some renaming to simplify exposition). `Table_Scan` (Sequential Scan) includes a `key` field which identifies the key of its input relation (if one exists). `Ind_Scan` (Index scan) includes a `key` field and also identifies indexed attributes ($\texttt{att}_1$, …, $\texttt{att}_n$). `Sort` reorders its input relation by sorting on attributes $\texttt{att}_1^{D_1}$, …, $\texttt{att}_n^{D_n}$ such that each D_i is either "ASC" or "DES". `Unique` removes duplicate tuples with equivalent values for attributes, $\texttt{att}_1$, …, $\texttt{att}_n$ in a single pass over its input. (Thus, `Unique` is always preceded in a Postgres plan by `Sort`.)[6] `Group` performs two passes over its input, first inserting `Null` values between pairs of consecutive tuples with different values for attributes, $\texttt{att}_1$, …, $\texttt{att}_k$, and then applying functions $F_{k+1}, \ldots, F_n$ to the collection of values of attributes $\texttt{att}_{k+1}, \ldots, \texttt{att}_n$ respectively, for each set of tuples separated by `Nulls`.[7] (Thus, `Group` is always preceded in a Postgres plan by `Sort` with the exception as described for `Unique`). `Append` appends the relations produced by subplans $\texttt{plan}_1$, …, $\texttt{plan}_n$. `Hash` builds a hash table over its input using a predetermined hash function over attribute, `att`. `HJoin` (Hash join) performs a (non-order-preserving) simple hash equijoin ($\texttt{att}_1 = \texttt{att}_2$) with the relation produced by `left` as the *probe* relation, and the relation produced by `right` as the *build* relation. (Thus, `HJoin` is always preceded in a Postgres plan by `Hash` over the `right` subplan.) `Merge` performs a merge equijoin ($\texttt{att}_1 = \texttt{att}_2$) with the relation produced by `left` as the *outer* relation, and the relation produced by `right` as the *inner* relation. (Thus, `Merge` is always preceded in a Postgres plan by `Sort` being applied to both the `left` and `right` subplans, except when an input to `Merge` is a result of an index scan.) Finally, `NLJoin` (nested-loop join) performs a nested-loop join with join predicate, `pred` over its inputs with with the relation produced by `left` as the *outer* relation, and the relation produced by `right` as the *inner* relation. If attribute `index?` is set to `TRUE`, then the join is an indexed nested loop join. Finally, we have added `NOP` as a dummy plan operator that we temporarily make the root of a Postgres plan prior to its refinement, and whose purpose is revealed in Section 4.2.

4.2 A Plan Refinement Algorithm

Our plan refinement algorithm accepts a query plan tree generated by Postgres as input and produces as output, an equivalent plan tree with unnecessary `Sort` operators (used either to order or group) removed. This algorithm requires that 4 new attributes be associated with every node in a query plan tree, n:

[4] In Postgres, grouping is accomplished by sorting so in actuality, our algorithm only eliminates unnecessary sorting operations. We chose Postgres as our target query optimizer because it is open-source.

[5] Postgres set operations (`SetOp`) are relevant but not yet covered by our refinement algorithm.

[6] Actually, this is not entirely accurate. `Sort` will not precede `Unique` if `Merge` precedes `Unique` and produces output in the required order.

[7] Postgres actually factors this operators into two single-pass operators: `Group` which inserts nulls between consecutive tuples with differing values for grouping attributes, and `Att` which aggregates on the resulting groups. As these operators are always consecutive in a plan, we collapse them into a single operator here.

Rule	*Join*	*Order Property Inference Rule*
1.	Nested Loop ($\bowtie^n$)	$O_o(R) \Rightarrow O_o(R \bowtie^n_p S)$
2.	Simple Hash ($\bowtie^{simh}$)	$O_o(R) \Rightarrow O_o(R \bowtie^{simh}_p S)$
3.	Nested Loop ($\bowtie^n$)	$O_o(R), Key_X(R), O_{o'}(S) \Rightarrow O_{o \rightarrow X^G \rightarrow o'}(R \bowtie^n_p S)$
4.	Merge ($\bowtie^m$)	$O_{A^O \rightarrow o}(R), Key_X(R), O_{B^O \rightarrow o'}(S) \Rightarrow O_{A^O \rightarrow o \rightarrow X^G \rightarrow B^O \rightarrow o'}(R \bowtie^m_{A=B} S)$
5.	Order-Preserving Hash ($\bowtie^h$)	Let $o = o_\alpha \rightarrow o_\beta$ s.t. $o_\alpha = A^O_1 \rightarrow \ldots \rightarrow A^O_{k-1}$ and $o_\beta = \bot$ or $o_\beta = A^G_k \rightarrow o_\gamma$. Then, $O_{o_\alpha \rightarrow o_\beta}(R), Key_X(R), O_{o'}(S) \Rightarrow O_{o_\alpha \rightarrow X^G \rightarrow o'}(R \bowtie^h_p S)$

Table 1: Order Property Inference Rules for Joins

Operator	*Attributes*
`Table_Scan`	`key`
`Ind_Scan`	`key`, $\texttt{att}_1$, ..., $\texttt{att}_n$
`Sort`	$\texttt{att}_1^{D_1}$, ..., $\texttt{att}_n^{D_n}$
`Unique`	$\texttt{att}_1$, ..., $\texttt{att}_n$
`Group`	$\texttt{att}_1$, ..., $\texttt{att}_k$, $\texttt{att}_{k+1}^{F_{k+1}}$, ..., $\texttt{att}_n^{F_n}$
`Append`	$\texttt{child}_1$, ..., $\texttt{child}_n$
`Hash`	`att`
`HJoin`	$\texttt{att}_1$, $\texttt{att}_2$
`Merge`	$\texttt{att}_1$, $\texttt{att}_2$
`NLJoin`	`pred`, `index?`
`NOP`	-

Table 2: Postgres Operators and Relevant Attributes

- `keys`: a set of attribute sets that are guaranteed to be keys of *inputs* to n;
- `fds`: a set of functional dependencies (attribute sets → attribute) that are guaranteed to hold of *inputs* to n;
- `req`: a single order property that is required to hold of *inputs* either to n or some ancestor node of n for that node to execute; and
- `sat`: a set of order properties that are guaranteed to be satisfied by *outputs* of n.

The basic idea of the plan refinement algorithm is to decorate the input plan with the attributes above, and then to remove any `Sort` operator `n` whose child node produces a result that is guaranteed to satisfy an order property required by its parent node (i.e., when `n.req` $\in$ `n.left.sat`). This is accomplished with 3 passes over the input plan as we discuss below, and illustrate with the refinement of the query plan of Figure 5a into the query plan of Figure 5c.

Pass 1: Functional Dependencies and Keys

A bottom-up pass is made of the tree so that functional dependencies (`fds`) and keys (`keys`) are propagated upwards when inferred to hold of intermediate query results. Keys and functional dependencies known of base relations are first used to decorate `Ind_Scan` and `Table_Scan` leaf nodes. These decorations propagate through most nodes unchanged, except through joins (`Merge`, `NLJoin` and `HJoin`) and `Unique` where new functional dependencies and keys are added, and `Append` where functional dependencies and key properties are lost. The functional dependencies and keys created by joins are those resulting from application of Axioms 3.1 and 3.2. `Unique` ($\texttt{att}_1$, ..., $\texttt{att}_n$) adds {$\texttt{att}_1$, ..., $\texttt{att}_n$} as an additional key, and {$\texttt{att}_1$, ..., $\texttt{att}_n$} → `targetlist` as an additional functional dependency.

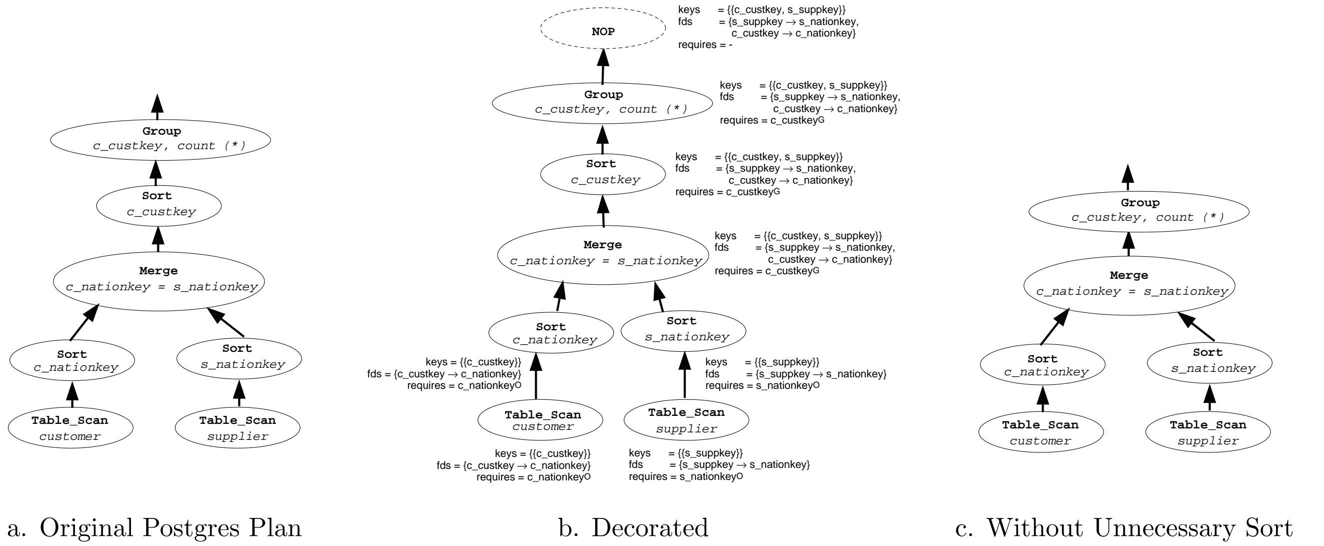
Figure 5b shows the decorated version of the plan of Figure 5a. The keys for base relations `Supplier` and `Customer` (`s_suppkey` and `c_custkey` respectively) propagate through their associated `Sort` nodes, as do the functional dependencies implied by these keys.[8] `Merge` creates a key which is the union of the keys of its inputs, and preserves both functional dependencies that hold of its inputs. The key and functional dependencies generated by `Merge` then propagate up to the root of the tree.

Pass 2: Required Order Properties

Next, a top-down pass is made so that required order properties (`req`) are propagated downwards from the root of the tree. The operation of this pass is captured by the pseudocode for `SetReq` shown in Figure 6. This algorithm is called on the root of the plan (`NOP`) with the empty order property to trigger the top-down pass. Observe that new required order properties are generated by:

- `NOP` (if its child is a `Sort` operator (i.e., if the original query includes an `Order By` clause),
- `Group` and `Unique` (which require inputs to be grouped on the grouping attributes),
- Join operators, each of which splits any required order property it inherits into separate required order properties for its child nodes according to the rules of Table 1.

[8] Because Postgres performs eager projection, only attributes `s_suppkey` and `s_nationkey` are output by the scan of `Supplier`. (Similarly, `c_custkey` and `c_nationkey` for `Customer`.)



a. Original Postgres Plan   b. Decorated   c. Without Unnecessary Sort

Figure 5: Query Plans for the Query of Figure 1

All other nodes pass the required order properties they inherit from parent nodes to their child nodes, except for `Hash` and `Append` which propagate the empty order property to their child nodes.

To illustrate, again consider the query plan shown in Figure 5b. The result of this query does not need to be ordered; hence, the `req` property of the `NOP` node is set to $\perp$. However, `Group` (which groups by `c_custkey`) requires its input be grouped by this attribute ($\texttt{c_custkey}^G$). This property gets pushed down to `Sort` and then `Merge`. Because this order property is not of the form, $o \rightarrow X^G \rightarrow o'$, order properties based on join attributes ($\texttt{c_nationkey}^O$ and $\texttt{s_nationkey}^O$) are passed down to the child nodes of `Merge`. These in turn are passed down to the `TableScan` nodes at the leaves of the tree.

Pass 3: Sort Elimination

The final pass of the plan refinement algorithm is a bottom-up pass of the query plan tree that determines what order properties are guaranteed to be satisfied by outputs of each node (`sat`), and that concurrently removes any `Sort` operator, `n` for which $\texttt{n.left.sat} \in \texttt{n.req}$. The operation of this pass is captured by the pseudocode for `InferSat` shown in Figure 7. Observe that new order properties are inserted into the `sat` set for the following nodes:

- `Ind_Scan` and `Sort` produce outputs that satisfy the order property, $\texttt{att}_1^{\texttt{O}} \rightarrow \ldots \rightarrow \texttt{att}_{\texttt{n}}^{\texttt{O}}$ (where $\texttt{att}_1, \ldots, \texttt{att}_{\texttt{n}}$ are the index and sort attributes respectively).

- `Table_Scan` produces an output that satisfies the order properties of the input relation it scans.

```
Algorithm SetReq (Node n, Order Property p)
 CASE n of
  NOP:            IF n.left = Sort THEN
                    n.req := n.left.att_1^O → ... → n.left.att_n^O
                  SetReq (n.left, n.req)
  Group, Unique:  n.req := n.att_1^G → ... → n.att_n^G
                  SetReq (n.left, n.req)
  Merge:          n.req := p
                  IF p = n.att_1^O → o → X^G → n.att_2^O → o' AND
                     X ∈ n.keys THEN
                    SetReq (n.left, n.att_1^O → o)
                    SetReq (n.right, n.att_1^O → o')
                  ELSE
                    SetReq (n.left, n.att_1^O)
                    SetReq (n.right, n.att_2^O)
  HJoin, NLJoin:  n.req := p
                  IF p = o → X^G → o' AND X ∈ n.keys THEN
                    SetReq (n.left, o)
                    SetReq (n.right, o')
                  ELSE
                    SetReq (n.left, p)
                    SetReq (n.right, ⊥)
  Hash:           n.req := p
                  SetReq (n.left, ⊥)
  Append:         n.req := p
                  SetReq (n.child_1, ⊥)
                  ...
                  SetReq (n.child_n, ⊥)
  Otherwise:      n.req := p
                  SetReq (n.left, p)
END
```

Figure 6: Pseudocode for `SetReq` (Pass #2)

- `Unique` and `Group` produce outputs that satisfy the order property, $\texttt{att}_1^{\texttt{G}} \rightarrow \ldots \rightarrow \texttt{att}_{\texttt{n}}^{\texttt{G}}$ (where $\texttt{att}_1, \ldots, \texttt{att}_{\texttt{n}}$ are the grouping attributes).

- Join operators produce outputs that satisfy the

order properties according to the rules of Table 1, and

- Append and Hash produce outputs that cannot be guaranteed to satisfy any order property.

All other nodes propagate the order properties they received from their child nodes to their parent nodes.

To illustrate, again consider the query plan of Figure 5b. During sort elimination, order properties that are guaranteed to be satisfied by inputs to every node are determined in a bottom-up pass of the query plan tree. Because s_suppkey is a key for Supplier, $\texttt{s_suppkey}^G$ is inferred to be satisfied by supplier. Similarly, $\texttt{c_custkey}^G$ is inferred to be satisfied by customer. Both of these order properties are still satisfied after application of Sort, though obviously Sort adds additional order properties ($\texttt{c_nationkey}^O$ and $\texttt{s_nationkey}^O$ for the left and right Sort nodes respectively). As a result of Merge, order properties: $\texttt{c_custkey}^{G} \rightarrow \texttt{s_suppkey}^{G}$, $\texttt{c_nationkey}^{O} \rightarrow \texttt{c_custkey}^{G} \rightarrow \texttt{s_suppkey}^{G}$, and $\texttt{c_custkey}^G$ are satisfied. Because the subsequent Sort has one of these order properties as its required order property ($\texttt{c_custkey}^G$), it can be removed as this order property is satisfied without sorting. This leaves the plan shown in Figure 5c.

Because of space limitations, we have omitted details regarding how the identities of Figure 4 are used to augment the order properties contained in a node's sat set (for example, adding $\texttt{c_nationkey}^G$ to the left Sort node of Figure 5b by application of Identity #2). In short, the identities of Figure 4 are applied lazily: only for nodes that are child nodes of Sort when the Sort is examined to see if it is unnecessary. We leave further details to our technical report [11].

4.2.1 Another Example: TPC-H Query 3

In [8], Simmen et al. used query 3 of the TPC-D (now TPC-H) benchmark to illustrate their order optimization techniques. This query returns the shipping priority and potential revenue of orders with maximum revenue of those not shipped as of some date, and is listed as it is in [8] (except with the date constant changed) below:

```
select l_orderkey,
   sum (l_extendedprice * (1 - l_discount)) as rev,
   o_orderdate, o_shippriority
from customer, orders, lineitem
where o_orderkey = l_orderkey
 and  c_custkey = o_custkey
 and  c_mktsegment = 'building'
 and  o_orderdate < date ('1998-11-30')
 and  l_shipdate > date ('1998-11-30')
group by l_orderkey, o_orderdate, o_shippriority
order by rev desc, o_orderdate
```

```
Algorithm InferSat (Plan p)
 Do a Bottom-Up (Preorder) Traversal of p.
 For each node, n:
  CASE n of
   Ind_Scan:
     n.sat := {n.att_1^O → ... → n.att_n^O}
   Table_Scan:
     n.sat := {Orders satisfied by input relation}
   Unique, Group:
     n.sat := n.left.sat ∪ {n.att_1^G → ... → n.att_n^G}
   Merge, NLJoin, HJoin:
     n.sat :=
      {o → X^G → o' |
        o ∈ n.left.sat, o' ∈ n.right.sat, X ∈ n.keys} ∪
      {o | o ∈ n.left.sat}
   Append, Hash:
     n.sat := {}
   Sort:
     n.sat := {n.att_1^O → ... → n.att_n^O}
     IF n.req ∈ n.left.sat THEN DELETE n
   Otherwise:
     n.sat := n.left.sat
END
```

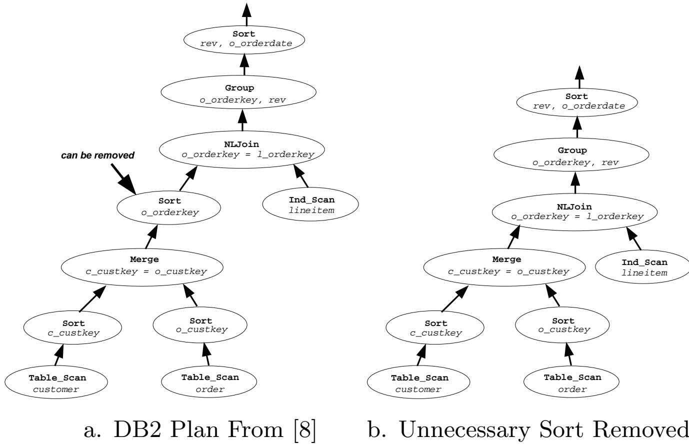
Figure 7: Pseudocode for InferSat (Pass #3)

Simmen et al. showed the plan produced for this query by DB2 without their order optimizations applied, and then showed the refined version of this plan (with a Sort operator removed) that was produced by applying their order optimizations. They showed that the optimized plan outperformed the original plan by a factor of 2. The optimized plan is reproduced from their paper in Figure 8a.

Because our approach to plan refinement reasons about groupings and secondary orderings, we can further improve the plan shown in Figure 8a by eliminating the Sort that precedes the NLJoin leaving the plan shown in Figure 8b. The Sort was introduced in [8] as an "early sort" optimization – it ensured that the output of the nested loop join was ordered on o_orderkey without having to sort the much larger relation output from the join. As with the previous example, the goal was to sort prior to aggregation (Group) so that aggregation could be performed in a single pass. The requirement that the input to aggregation be sorted on o_orderkey is stronger than necessary. In fact, the input need only be grouped, and the output of nested loop is guaranteed to be grouped on o_orderkey even without sorting it.

Due to space considerations, we illustrate the reasoning that permits refinement of this plan at a high-level only:

1. By the definition of sorting, the output of the sorts of Customer and Order are R and S such that $O_{c_custkey^O}(R)$ and $O_{o_custkey^O}(S)$. By identity #2 of Figure 4, we also have $O_{c_custkey^G}(R)$ and $O_{o_custkey^G}(S)$.



a. DB2 Plan From [8] b. Unnecessary Sort Removed

Figure 8: TPC-H Q3: Effects of Plan Refinement

2. By identity #5 of Figure 4, every group of `Order` tuples with the same value of `o_custkey` trivially satisfies the order property, $\texttt{o_orderkey}^G$. Thus we have, $O_{\texttt{o_custkey}^G \rightarrow \texttt{o_orderkey}^G}(S)$.

3. By Rule 4 for `Merge` in Table 1, we have

$$O_{\texttt{c_custkey}^G \rightarrow \texttt{c_custkey}^G \rightarrow \texttt{o_custkey}^G \rightarrow \texttt{o_orderkey}^G}(T),$$

for join result, T. Note that the join predicate ensures that `c_custkey = o_custkey` for all output tuples, so by identity #1 of Figure 4, this becomes:

$$O_{\texttt{o_custkey}^G \rightarrow \texttt{o_custkey}^G \rightarrow \texttt{o_custkey}^G \rightarrow \texttt{o_orderkey}^G}(T).$$

4. Because `o_custkey` functionally determines itself, we can apply identity #4 of Figure 4 twice to get, $O_{\texttt{o_custkey}^G \rightarrow \texttt{o_orderkey}^G}(T)$.

5. By identity #5 of Figure 4, because `o_orderkey` (as the key of `Order`) functionally determines `o_custkey`, we have $O_{\texttt{o_orderkey}^G}(T)$.

6. By Rule 1 of Table 1 for `NLJoin`, we have $O_{\texttt{o_orderkey}^G}(U)$ for the result of the nested loop join, U.

Thus, the result of the nested loop join is grouped on `o_orderkey`, even without a prior sort on `o_orderkey`, and there is no reason to sort on this attribute.

5 Results

To measure the degree to which our plan refinement techniques improve query processing performance, we ran two experiments based on the example plan refinements described in Section 4.2.1. Both experiments were run with Postgres after we modified the optimizer code to implement the plan refinement algorithm described in Section 4.2. We describe these two experiments in Sections 5.1 and 5.2 respectively, and in Section 5.3, report on an experiment determining the cost of performing the plan refinement optimization.

We ran all of our experiments on a 1 Ghz Pentium III running Linux, with 512 MB RAM and a 120 GB hard disk. Each experiment was run 3 times with average times reported. Timing results for plan executions were generated by the Postgres EXPLAIN ANALYZE tool, which also provided details about timings for individual operators, sizes of intermediate query results etc. Our data was generated by the DBGEN tool for TPC-H with a scale factor of 1. Thus, each query ran on a `Customer` table containing 150,000 rows, a `Supplier` table containing 10,000 rows, an `Order` table containing 1,500,000 rows, and a `LineItem` table with 6,000,000 rows.

5.1 *Experiment #1: Figure 1*

Our first experiment involved comparing the performance of the Postgres plan generated for the query of Figure 1a (Figure 5a), and the plan returned by our plan refinement algorithm (Figure 5c). The results are shown in the table below.

Postgres Plan (Fig. 5a)	Refined (Fig. 5c)	Ratio
6384.9 sec	487.9 sec	13.08

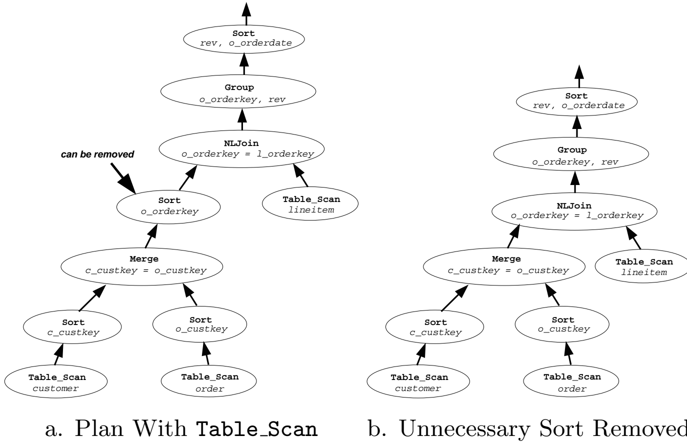
It should be pointed out that this query joined `Customer` and `Supplier` on non-key attributes with a very small range of values (`nation_key`). Therefore, the output of the join was extremely large (close to 60 million rows), thereby making the plan that performed a sort before aggregating become bogged down despite fairly small input relations. This is an extreme example of when it pays to avoid sorting: when it is done towards the end of the computation on intermediate join results where the join selectivity is very low. In such cases, plan refinement can reduce execution costs by an order of magnitude.

5.2 *Experiment #2: TPC-H Query 3*

Our second experiment involved comparing plans for TPC-H query 3 shown in Figures 8a (produced by DB2 as shown in [8]) and 8b (the same plan with the unnecessary `Sort` removed). The timing results for each of these plans is shown below.

Plan From [8] (Fig. 8a)	Refined (Fig. 8b)	Ratio
126.8 sec	2729.9 sec	0.05

Interestingly enough, removing the `Sort` from the original plan dramatically **increased** the execution time of the plan! This was due to the effect of sorting prior to an indexed nested loop join on caching behavior. Specifically, sorting the outer relation of the join on the join attribute (`o_orderkey`) had the effect of ensuring



a. Plan With Table_Scan   b. Unnecessary Sort Removed

Figure 9: TPC-H Q3 W/ Table_Scan

that index lookups for the same value of o_orderkey were consecutive, thereby increasing the likelihood of finding joining tuples from lineitem in the cache. Further, because o_orderkey was not only grouped but sorted, consecutive tuples with differing values of o_orderkey were likely to have values of o_orderkey that were close, thereby increasing the likelihood that index nodes that were accessed during the index scan would also be found in the cache.

To test our theory regarding this result, we modified the query plan produced by DB2 to perform a Table_Scan of lineitem (rather than an Ind_Scan), as would be required if there were no index on l_orderkey over lineitem. This plan is shown in Figure 9a. The refined plan with the unnecessary Sort operator removed is shown in Figure 9b, and the timing results for these two plans is shown below:

TPC-H Q3 With Table_Scan (Fig. 9a)	Refined (Fig. 9b)	Ratio
121.4 sec	113.3 sec	1.07

Note that not only does removal of the Sort operator make the above plan execute 7% faster, the resulting plan also executes 12% faster than the original plan produced by DB2's order optimization that uses indexed nested loop join rather than nested loop join! Again, this shows the potential performance gains that can result from refining plans to remove unnecessary sorting and grouping.

5.3 *Experiment #3: Overhead*

For our final experiment, we measured the overhead added to the cost of query optimization resulting from performing plan refinement. For each of the experiments above, we measured the times required for Postgres to optimize its queries both with and without our plan refinement extension. We also measured these

Query	Opt Time No Refinement	Cost Of Refinement	Added Overhead
Fig. 5a	0.56 msec	0.11 msec	19.6%
Fig. 8a	2.85 msec	0.32 msec	11.2%
TPC-H Q1	3.54 msec	0.06 msec	1.7%
TPC-H Q5	46.07 msec	1.39 msec	3.0%
TPC-H Q10	52.10 msec	0.33 msec	0.6%

Table 3: Overhead Of Plan Refinement

times for some TPC-H queries that are *not* changed as a result of executing plan refinement algorithm: TPC-H query #'s 1, 5 and 10. The results are shown in Table 3. Interestingly, the overhead introduced by plan refinement was only significant (i.e., 5% or more) when refinement actually had an effect! And as we saw earlier, the overhead introduced is significantly outweighed by the savings in query execution cost in most cases. For all TPC-H queries that are unaffected by plan refinement, the overhead introduced by plan refinement is negligible.

6 Related Work

Related work in avoiding sorting and grouping tends to fall in one of two camps: either it is concerned with inferring orderings or with inferring groupings. Ours is the first paper to our knowledge to fully integrate the two.

The earliest work on order optimization was from System R [6]. System R kept track of orderings known of intermediate query results so as to potentially influence the choice of join strategy (inferred, or "interesting orders") might influence a selection of a sort-merge join processing strategy if the inferred ordering made it possible to avoid one or more of the sorts. While pioneering in this area, the techniques used for inferring orderings were quite primitive (based on explicit ORDER BY clauses). Grouping inference and secondary orderings were not considered.

The seminal work of Simmen, Shekita and Malkemus [8] introduced the use of functional dependencies to infer orderings of join results. However, they consider only primary orderings and not secondary orderings nor grouping. Therefore their techniques would fail to optimize the query plans we showed in Figures 5a and 8a. In fact, the latter plan was taken from their paper where they presented it as the final product of their order optimization techniques.

Slivinskas, Jensen and Snodgrass [9] also contributed work on ordering, though less in the context of query plan generation but more at the level of the data model. They propose three different notions of relation equivalence (list-based, multiset-based and set-based) corresponding to the three ways that SQL treats a relation depending on the query. They point out that each defines a different class of acceptable query results. They discuss order preserving operators such as

nested loop join and selection, but in the context of how they lead to list-equivalence. They do not consider how grouping properties get propagated.

Less work can be found on inference of grouping properties, though the most notable work is that of Chaudhuri and Shim [2] and Yan and Larsen [12]. The goal in both cases is to develop "early group-by" optimization strategies (analagous to the early sorting goals of [8] and [5]. In so doing, they approach the inference of grouping in a manner that is the reverse of our approach: whereas we move from the bottom of a plan to the top to see what (grouping or sorting) operations can be removed, they move from the top of a plan to the bottom to see what grouping operations can be pushed down. Both approaches require understanding of how grouping properties get propagated by operators.

7 Conclusions

In this paper, we present a formal and approach to order optimization that integrates both orderings and groupings within the same comprehensive framework. We make the following novel contributions:

1. Aside from orderings, we also infer how relations are *grouped*. Just as inference of orderings can make it possible to avoid sorting, inference of groupings can make it possible to avoid hash-based grouping algorithms. We also consider *secondary* orderings and groupings. By inferring secondary orderings and groupings, it is possible to avoid unnecessary sorting or grouping over multiple attributes. Also, in some cases one can use secondary orderings known of an operator's input to infer primary orderings of of its output.

2. We present a formal semantics for order properties, thereby making it easier to reason about and verify inference techniques.

3. We introduce a *plan refinement* algorithm that decorates query plans produced by Postgres with inferred orderings and groupings and then refines these plans by removing unnecessary sorting and grouping operations.

4. We empirically show the benefits of plan refinement and the low overhead it adds to the cost of query optimization.

One topic for future work concerns the integration of the plan refinement with the plan generator of a query optimizer, so that plans with "interesting orders" and "interesting groupings" might be retained as candidate plans, where they might currently be discarded. We also are interested in generalizing this work to infer "bounded disorder": unordered relations whose disorder can be measured as the number of passes of a bubble sort required to make the relation ordered. Inference of "bounded disorder" appears to be relevant when considering how order properties get propagated through block-nested-loop joins, and could be exploited to reduce the cost of certain plan operators.

8 Acknowledgements

We would like to thank Dapeng Xu for his suggestions concerning the formalization of order properties, and Dave Maier for suggesting consideration of groupings. We would also like to thank Joe Hellerstein for providing his expertise on the Postgres query optimizer.

References

[1] Kjell Bratbergsengen. Hashing methods and relational algebra operations. In Umeshwar Dayal, Gunter Schlageter, and Lim Huat Seng, editors, *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*, pages 323–333. Morgan Kaufmann, 1984.

[2] Surajit Chaudhuri and Kyuseok Shim. Including Group-By in Query Optimization. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 354–366, Santiago, Chile, 1994.

[3] J. Claussen, A. Kemper, D. Kossmann, and C. Wiesner. Exploiting early sorting and early partitioning for decision support query processing. *VLDB Journal: Very Large Data Bases*, 9(3):190–213, 2000.

[4] H. Darwen and C. Date. The role of functional dependencies in query decomposition. In *Relational Database Writings 1989-1991*. Addison Wesley, 1992.

[5] D. Kossmann J. Claussen, A. Kemper. Order-preserving hash joins: Sorting (almost) for free. 1998.

[6] P. Griffiths Selinger, M.M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 23–34, 1979.

[7] Abraham Silberschatz, Henry F. Korth, and S. Sudershan. *Database System Concepts, 4th edition*. Computer Science Series. McGraw-Hill, 2001.

[8] David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, Montréal, Québec, Canada, June 1996.

[9] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Bringing order to query optimization. *ACM SIGMOD Record*, 31(2):5–14, 2002.

[10] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The implementation of postgres. *Transactions in Knowledge and Data Engineering*, 2(1):125–142, 1990.

[11] Xiaoyu Wang and Mitch Cherniack. Avoiding ordering and grouping during query processing. Brandeis Technical Report, June 2003.

[12] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, pages 89–100. IEEE Computer Society Press, 1994.