Slicing Long-Running Queries

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Ravi Ramamurthy
Microsoft Research
ravirama@microsoft.com

ABSTRACT

The ability to decompose a complex, long-running query into simpler queries that produce the same result is useful for many scenarios, such as admission control, resource management, fault tolerance, and load balancing. In this paper we propose query slicing as a novel mechanism to do such decomposition. We study different ways to extend a traditional query optimizer to enable query slicing and experimentally evaluate the benefits of each approach.

1. INTRODUCTION

New application scenarios have significantly increased the complexity of queries that are submitted to a database server. In this context, it is common for queries to run for a long time and consume significant server resources. These long-running queries, in turn, introduce new challenges to administer and tune the underlying database system, as illustrated by the following examples:

Admission control: Many systems rely on strict admission control policies to prevent long-running queries from monopolizing system resources. In such systems, a query is accepted only if its estimated cost is below a threshold. Examples include traditional database systems [11] as well as emerging cloud data services [14]. Although such limits appear restrictive, they are necessary to ensure the overall scalability and performance of the shared infrastructure for all users. No matter what threshold is used for admission control, however, there will still be valid queries that are too expensive to run completely. In such systems, application developers need to manually transform a query that is not admitted into simpler queries that individually pass the admission test.

Resource management: In addition to admission control, an important component of resource management is scheduling, which maintains and manages a queue of pending tasks [9, 10]. Designing robust resource management policies in the presence of multiple long-running queries remains a challenging task. For instance, techniques that abort a long-running query in favor of another with higher priority face the challenge of restarting the aborted query from scratch, potentially wasting considerable work. Pause/restart techniques [3, 4] partially deal with these issues, but do not handle all scenarios gracefully. The ability to decompose a complex query into simpler fragments can be an important step in addressing resource management challenges.

Fault tolerance: Conceptually similar to the case of manually aborted queries, a long-running query that fails before completion has to be restarted from scratch [3, 4, 16]. If a query is decomposed into simpler components, these can be restarted at a finer granularity, thus minimizing the amount of wasted computation.

Load Balancing: Parallel systems attempt to distribute computation across nodes in such a way that each node performs roughly the same amount of work. This task becomes more challenging when the units of distribution are long-running and complex. The ability to decompose a long-running query into many pieces of similar cost can thus contribute in adapting load balancing techniques to new scenarios.

In this paper we propose *query slicing* as a novel mechanism to complement existing work in the context of managing long-running queries. The idea is to enable a query optimizer to decompose a complex query into *slices* that are executed to produce the original result. Specifically, in this paper we study the following version of the query slicing problem. For an input query q and a given cost threshold, we attempt to decompose q into a set of queries $\{q_i\}$ such that (i) all $\{q_i\}$ together produce the original result, and (ii) the cost of each individual query is bounded by the cost threshold (we formally define the problem in Section 2). Consider the following query q:

```
q =  SELECT R.a, S.b
     FROM R JOIN S ON R.x=S.y
     WHERE R.c<10 AND S.d>20
```

Suppose that the cost threshold for a slice is smaller than the original cost of the query. In this case, if `R` is a large table and `R.c<10` returns a small fraction of `R`, q can be rewritten as two queries q_1 and q_2 as shown in Figure 1. Although the combined cost of q_1 and q_2 is larger than that of q due to an intermediate table creation, each q_1 and q_2 might individually satisfy the cost threshold. While this extension to traditional query optimization seems natural, there are significant challenges in implementing such functionality. For example, even for such a simple query, there can be several other alternatives to consider. Suppose that there is an index on `S.d`. In that case, we can partition `S` into two fragments by adding predicates on `S.d` and rewrite q into q_3 and q_4 as shown in Figure 1. In this case, query q can be decomposed into two slices q_3 and q_4 that can be efficiently executed using index-based plans. As a final example, suppose that both `R.c<10` and `S.d>20` are not very selective (i.e., they return most of `R` and `S` respectively). If there are indexes on `R.x` and `S.y`, another alternative to evaluate q is given by q_5 and

Proceedings of the VLDB Endowment, Vol. 3, No. 1

```
q1 = INSERT INTO TR
     SELECT R.a, R.x
     FROM R
     WHERE R.c<10

q2 = SELECT TR.a, S.b
     FROM TR JOIN S ON TR.x=S.y
     WHERE S.d>20
```

```
q3 = SELECT R.a, S.b
     FROM R JOIN S ON R.x=S.y
     WHERE R.c<10 AND S.d>20
                  AND S.d < 100

q4 = SELECT R.a, S.b
     FROM R JOIN S ON R.x=S.y
     WHERE R.c<10 AND S.d>=100
```

```
q5 = SELECT R.a, S.b
     FROM R JOIN S ON R.x=S.y
     WHERE R.c<10 AND S.d>20 AND R.x<500

q6 = SELECT R.a, S.b
     FROM R JOIN S ON R.x=S.y
     WHERE R.c<10 AND S.d>20 AND R.x>=500
```

Figure 1: Different ways to decompose an input query into two slices.

q_6 in Figure 1. In this case, q_5 and q_6 implement a "partitioned"-join strategy, and their results together are the same as those of q. Even when there are no indexes on `S.y`, if table `S` is small, q_5 and q_6 would each join a fragment of `R` with the whole `S`, producing results efficiently.

The examples above illustrate that there can be multiple ways to decompose a query into components that satisfy the cost threshold and together produce the same original result. In this paper we introduce a comprehensive approach to decompose such long-running queries into multiple *slices*, such that each slice satisfies a cost threshold and the global execution is as efficient as possible. The rest of the paper is structured as follows. In Section 2 we formalize our problem statement and the minor extensions to an execution engine that are required for our techniques. In Section 3 we present a family of optimization strategies that tradeoff optimization time and quality of the resulting solutions. In Section 4 we report an experimental evaluation of our approaches. Finally, in Section 5 we review related work.

2. QUERY SLICING

In this paper we consider `SQL` queries and the optimizer's cost model as the estimator for query costs. We then state the *query slicing* problem as follows. Let *cost(q)* be the optimizer's estimated cost for query q, and Δ the cost threshold for any query slice (note that if $\Delta \geq cost(q)$, the original plan is optimal). Slicing q for Δ produces a partially ordered set of queries $\{q_1, \ldots, q_n\}$ such that:

1. Executing all q_i (while respecting the partial order) produces a table containing the same result[1] as q.
2. $\forall_i$ *cost*$(q_i) \leq \Delta$.
3. $\sum_i$ *cost*(q_i) is minimal.

We require that the final result be written into a table, which can then be read by the user at any time. Otherwise, any query slice that involves memory intensive operators like hash joins, could be opened by the client and processed very slowly, using significant server resources. This requirement does not affect our algorithms, which can be easily adapted to stream results of such query slices to the client without the last materialization.

In Figure 1 we showed different ways to slice queries, which include writing intermediate results into temporary tables and horizontally partitioning the input tables. We next formalize these alternatives using the notion of *extended execution plans*.

2.1 Extended Execution Plans

Extended execution plans enable reasoning with collections of query slices very similarly to what is done with a traditional query, thus leveraging existing work in query optimization. In addition to the traditional relational operators, extended execution plans can contain *partitioned spools*. Partitioned spools are a useful formalism to reason with query slices, are expressive enough to handle scenarios including those in Section 1, and can be implemented in existing system with minimal or even no changes at all. We next describe different variants of the partitioned spool operator.

The Spool Operator: The *Spool* operator χ (used in almost all DBMS engines) writes an intermediate result into a temporary table. It takes a single relational input R and a temporary table name, and bulk-loads the temporary table with the result of evaluating R. *Spool* operators can be placed on top of any execution sub-plan, and the resulting temporary table can be subsequently read in an extended execution plan. If so, we connect the *Spool* operator χ and the consumer scan with a dotted line. Queries q_1 and q_2 in Section 1 can be implemented using a *Spool* operator as shown in Figure 2(a). The scan operator above the *Spool* operator reads from the temporary table, called `TR` (omitted when it is clear in context).

The Input-Partitioned Spool Operator: The *input-partitioned spool operator*, or *iSpool* for short, extends the *Spool* operator by introducing iteration. The relational input R of an *iSpool* operator is parameterized by a predicate of the form $\$l<c\leq\$h$, where c is a column defined in R. It additionally defines an expression of the form $(c, \{r_1, r_2, \ldots, r_k\})$, where r_i=$(l_i, h_i]$ are ranges that form a partition of c's domain. To process an *iSpool* operator $\chi_{(c,\{r_i\})}(R)$ we instantiate R for each range $l_i < c \leq h_i$, denoted by $R[l_i, h_i]$, and evaluate $\chi(R[l_i, h_i])$). Note that the *Spool* operator appends the results of each iteration to the same temporary table. In an extended execution plan, we mark with double lines the edges that vary for each instantiated range. An extended execution plan has double lines for all operators in the path connecting the *iSpool* operator to the base tables over which the range column is defined (there might be multiple such tables due to join predicates), unless the path includes another *Spool* operator. Queries q_3 and q_4 in Section 1 are implemented using *iSpool* operators in Figure 2(b).

The Output-Partitioned Spool Operator: The *iSpool* operator iterates over multiple relations and produces a single temporary output table. Conversely, the output-partitioned spool operator, or *oSpool* for short, takes a single input relation and partitions it into multiple temporary output tables. As with *iSpool*, an *oSpool* operator takes a parameter $(c, \{r_1, r_2, \ldots, r_k\})$, where c is a column of the *oSpool*'s relational input and $\{r_i\}$ forms a partition of c's domain. To process an *oSpool* operator $\chi^{(c,\{r_i\})}(R)$ we maintain as many temporary tables as ranges in the operator [2]. We then read R completely and append each tuple to the appropriate table depending on the value of column c. The *oSpool* operator is similar to partitioning operators used in parallel databases, and we discuss this relationship in Section 5. We note that *oSpool* operators can be easily implemented by a small code fragment that leverages querying and bulk-loading capabilities of existing query engines. Figure 2(c) shows an extended execution plan that implements queries q_5 and q_6 with a partitioned join on `x` and `y` using an *iSpool* operator. The plan joins together tuples from `R` and `S` that satisfy each range of `R.x` (respectively `S.y` due to the join predicate). The valid tuples from `S` for each range are obtained by an index on `S.y`. Suppose, however,

[1] We assume that no updates occur across executions of q_i.

[2] *oSpool* arguments use superscripts and *iSpool* arguments use subscripts.

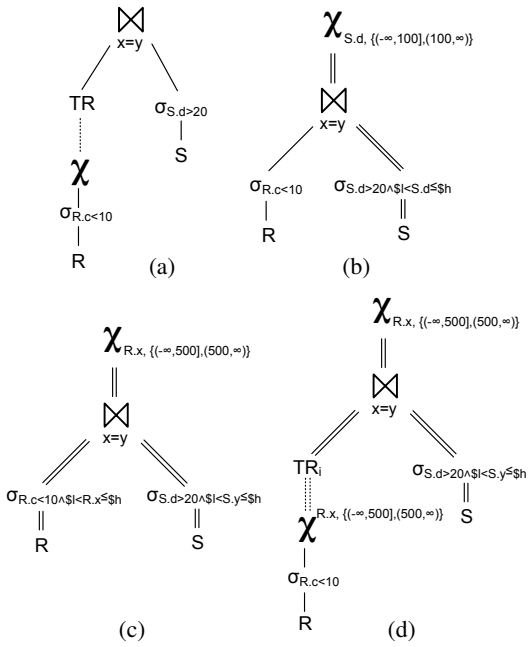


Figure 2: Extended execution plans to reason with query slices.

that there is no index on R.x, so processing $\sigma_{R.c<10 \wedge l_i<R.x\leq h_i}(R)$ for each range requires scanning the whole R. We can improve this plan by introducing an *oSpool* operator, which reads $\sigma_{R.c<10}(R)$ once and writes two temporary tables $\texttt{TR}_0$ and $\texttt{TR}_1$ depending on R.x values (see Figure 2(d)). These temporary tables contain tuples from R satisfying both R.c<10 and $l_i <$R.x$\leq h_i$ (i.e., the tuples needed for each iteration of the *iSpool*).

The Input/Output-Partitioned Spool Operator: Finally, the input/output-partitioned spool, or *ioSpool* for short, efficiently combines *iSpool* and *oSpool* while scanning the input data once. An *ioSpool* takes two expressions $(c_{in}, \{r_i\})$ and $(c_{out}, \{s_j\})$, and a relational input R parameterized by a range predicate on c_{in}. To process an *ioSpool* $\chi^{(c_{out},\{s_j\})}_{(c_{in},\{r_i\})}(R)$ we evaluate, for each range $l_i < r_i \leq h_i$, expression $\chi^{(c_{out},\{s_j\})}(\sigma_{c_{in}\in r_i}(R))$, where the *oSpool* operator shares the temporary tables across iterations. Suppose, in Figure 2(d), that evaluating $\sigma_{R.c<10}(R)$ is too expensive. By changing the *oSpool* in the figure to an *ioSpool* $\chi^{(R.x,\{r_i\})}_{(R.c,\{r_j\})}$, and assuming that an index on R.c is available, we obtain an admissible execution plan.

In this paper we focus on range partitions for simplicity, but our approach can be extended to consider hash partitioning as well.

2.2 Valid Extended Execution Plans

To evaluate an extended execution plan P, we first obtain query slices by *breaking-up* P on all dotted-line edges. Each query slice depends on base or intermediate tables, which induce a partial order among slices. We then execute query slices respecting this partial order. A valid extended execution plan satisfies some restrictions on the placement of spool operators. We say that an *oSpool* (or *ioSpool*) operator χ with output parameter $(c, \{r_i\})$ *closes* another *iSpool* (or *ioSpool*) operator χ' with input parameter $(c', \{s_j\})$ if χ is a descendant of χ', c=c' and $\{r_i\}$=$\{s_j\}$. For an extended execution plan to be valid, every time there is an *iSpool* (or *ioSpool*) operator χ with input parameter $(c, \{r_i\})$ and we follow the path from χ to the base table(s) that defines c (modulo column equivalence) the first spool operator in the path (if any) has to close χ.

2.3 Cost Model for Extended Execution Plans

The cost model in a traditional optimizer needs to be extended to reason with spool variants, query slices and cost thresholds. The *local cost* of an operator ρ, $LC(\rho)$, is given by traditional cost formulas of query optimizers (spool variants are seen as table insertions and thus costed appropriately). Additionally, we need to extend the cost model by defining, for each execution subplan, a tuple (SC, DC), where *SC* is the *shallow cost* of the subplan (which models the cost of a query slice and should fit in the cost threshold), and *DC* is the *deep cost* of the subplan (which should be minimized).

Consider a scan or a seek operator over a table in an execution plan. If ρ's table is a base table, we define $SC(\rho)$=$LC(\rho)$, and $DC(\rho)$=$LC(\rho)$. If ρ's table is a temporary result from executing subplan P, the scan operator *resets* the shallow cost *SC* of its subplan to its local cost, and the overall cost is kept in *DC*. That is, $SC(\rho)$=$LC(\rho)$, and $DC(\rho)$=$LC(\rho)$+$DC(P)$.

Consider an execution plan P with root operator ρ and subtrees $\rho_1, \ldots, \rho_n$. If ρ does not partition its input (i.e., ρ is not an *iSpool* or *ioSpool* operator), $SC(P)$=$LC(\rho) + \sum_i SC(\rho_i)$, and $DC(\rho)$=$LC(\rho) + \sum_i DC(\rho_i)$. Suppose now that ρ=$\chi_{(c,\{r_i\})}$ with an input parametric plan ρ' (the case for an *ioSpool* is defined analogously). Then, the shallow cost for ρ is the maximum, over all ranges r_i, of executing the parametric plan $\rho'[r_i]$ and writing the partial result to the temporary table. The deep cost for ρ is the sum of the local and deep costs for the first range r_1, and the local and shallow costs of subsequent ranges. The reason is that we only incur a deep cost once (to materialize intermediate results down in the execution plan) but subsequent iterations of the *iSpool* operator would read from the temporary tables, therefore incurring only the shallow cost (if ρ' has no spool operators, $DC(\rho'[r_i]) = SC(\rho'[r_i])$). More formally,

$$SC(\rho) = \max_i(LC(\chi(\rho')[r_i]) + SC(\rho'[r_i])$$

$$DC(\rho) = \sum_{i=1}^{n} LC(\chi(\rho')[r_i]) + DC(\rho'[r_1]) + \sum_{i=2}^{n} SC(\rho'[r_i])$$

We now reformulate the *query slicing* problem. Let q be a query and Δ be the cost threshold for any query slice. Slicing q for Δ produces an extended execution plan P so that (i) $SC(p) \leq \Delta$ for every subplan p of P, and (ii) $DC(P)$ is minimal.

3. FINDING OPTIMAL QUERY SLICES

In this section we introduce several optimization strategies to solve the query slicing problem. Our approach results in a spectrum of alternatives that balance optimization cost and quality of the resulting plans. We focus on SPJ queries, and extend the class of queries that we can handle in Appendix B. To explain our algorithms, we first show, in Figure 3, a simplified top-down[3] version of a dynamic programming algorithm that obtains the best execution plan for an SPJ query. We assume that a global Memo associative array is available, which takes a subset of tables $\mathcal{R}$ and a sort order $\mathcal{S}$, and returns the best plan for such combination $(\mathcal{R}, \mathcal{S})$.

The optimization of a query starts by calling optimize$(\mathcal{R}, null)$ or optimize$(\mathcal{R}, c)$ if an order by c is required due to an order-by clause. Line 1 implements memoization and calculates the best plan in lines 2-15 once for each distinct $(\mathcal{R}, \mathcal{S})$ (otherwise, it simply returns the cached version). To compute the best plan, lines 2-4 try to implement a candidate plan CP using an enforcer plan if an

[3]The top-down approach with on-demand interesting orders is very similar to the traditional bottom-up dynamic programming approach of System-R[13], but avoids explicitly enumerating all interesting orders upfront, or otherwise generating unneeded alternatives.

```
updateMemo (R:tables, S:order, P:plan)
01 if (P ≠ null and (Memo[R, S] = null or
                     cost(P) < cost(Memo[R,S]))
02    Memo[R, S] = P
```

```
optimize (R:tables, S:order)
returns best plan for R satisfying S
01 if (Memo[R, S] was not yet calculated)
02    if (S ≠ null)
03       CP = Sort_S(optimize(R, null))
04       updateMemo(R, S, CP)
05    if (|R| = 1)
06       CP = best single-table plan under S order
07       updateMemo(R, S, CP)
08    else for each valid partition (R1, R2) of R
09       for each join algorithm JA
10          S1,S2 = required orders of R1,R2 for JA
11          CP1 = optimize(R1, S1)
12          CP2 = optimize(R2, S2)
13          if (CP1 ≠ null and CP2 ≠ null)
14            CP = JA(CP1, CP2)
15            updateMemo(R, S, CP)
16 return Memo[R, S]
```

Figure 3: Top-down dynamic programming join reordering.

order is requested (i.e., $\mathcal{S} \neq$ `null`). In that case, line 3 recursively calculates the best plan for the same tables $\mathcal{R}$ without requesting any order, and inserts a top-most `sort` operator which would enforce the required order. Line 4 calls `updateMemo` with the resulting plan, which updates the best plan found so far for $(\mathcal{R}, \mathcal{S})$.

For any value of $\mathcal{S}$, lines 5-15 calculate the best plan satisfying the required sort order. Lines 5-7 handle the case of a single table in $\mathcal{R}$, obtain the best single-table plan satisfying order $\mathcal{S}$, and update the memo with such candidate plan. For the general case of $|\mathcal{R}|>1$ line 8 obtains all valid partitions of $\mathcal{R}$ into $\mathcal{R}1$ and $\mathcal{R}2$ (e.g., if only considering left-deep trees, the partitions must satisfy $|\mathcal{R}2|=1$). For each such partition and join algorithm `JA`, line 10 calculates the required orders of the join inputs (e.g., a merge join operator requires both inputs to be sorted on the respective join columns). Lines 11-12 recursively obtain the best plans for $\mathcal{R}1$ and $\mathcal{R}2$, and lines 14-15 assemble the join plan and update the memo. After all partitions and join alternatives have been evaluated, line 16 returns the actual content of `Memo[`$\mathcal{R}$`,`$\mathcal{S}$`]`, which contains the best plan for the input set of tables and required order.

3.1 Handling Spool Operators

We next describe a simple extension to the algorithm of Figure 3 that considers spool operators (Section 2.1). To that end, every time we create a candidate plan and call `updateMemo` in lines 4, 7, and 15, we additionally consider spooling such intermediate results by adding after line 15 (and also after 4 and 7):

```
15.1    updateMemo(R, S, Scan(Spool(CP)))
```

We also need to consider only valid execution plans (i.e., those that satisfy the cost threshold Δ). Thus, we modify the predicate `cost(P) < cost(Memo[`$\mathcal{R}$`,`$\mathcal{S}$`])` in line 1 of `updateMemo` as follows:

$$DC(\texttt{P}) < DC(\texttt{Memo}[\mathcal{R},\mathcal{S}]) \text{ and } \forall \texttt{p} \in \texttt{P}: SC(\texttt{p}) \leq \Delta$$

In other words, we reject plans that contain a subplan with shallow cost exceeding the threshold Δ, and keep the one with the smallest deep cost. These changes are necessary, but unfortunately not sufficient to obtain the optimal slicing strategy. Suppose, as a very simple example, that we call `optimize({R},null)` and that there is a single-table predicate `R.a` < 10 on table `R`. The algorithm would then generate the following two plans:

- P_1 = $\texttt{Filter}_{R.a<10}\texttt{(Scan(R))}$
- P_2 = $\texttt{Scan(Spool(Filter}_{R.a<10}\texttt{(Scan(R))))}$

```
updateMemo (R:tables, S:order, P:plan)
01 if (P ≠ null and ∀p ∈ P: SC(p)≤ Δ)
02    Memo[R, S] = skyline(Memo[R, S] ∪ P)
```

```
optimize-S (R:tables, S:order)
returns skyline of plans for R satisfying S
01 if (Memo[R, S] was not yet calculated)
02    if (S ≠ null)
03       for each (CP ∈ optimize-S(R, null))
04          updateMemo(R, S, Sort_S(CP))
05          updateMemo(R, S, Scan(Spool(Sort_S(CP))))
06    if (|R| = 1)
07       CP = best single-table plan under S order
08       updateMemo(R, S, CP)
09       updateMemo(R, S, Scan(Spool(CP)))
10    else for each valid partition (R1, R2) of R
11       for each join algorithm JA
12          S1,S2 = required orders of R1,R2 for JA
13          CP1 = optimize-S(R1, S1)
14          CP2 = optimize-S(R2, S2)
15          for each (pCP1, pCP2) ∈ CP1 × CP2
16            CP = JA(pCP1, pCP2)
17            updateMemo(R, S, CP)
18            updateMemo(R, S, Scan(Spool(CP)))
19 return Memo[R, S]
```

Figure 4: Handling *Spool* operators for query slicing.

Assume that $SC(P_1)$=$DC(P_1)$=100. Because the *Spool* operator only materializes the tuples that satisfy `R.a` < 10, and only the columns that are relevant upwards in the tree, the cost of reading the temporary table would be smaller than that of scanning the original table R. That is, it could be that $SC(P_2)$=20 and $DC(P_2)$=150. In this case, it is not clear which one among P_1 and P_2 we should keep in `Memo[{R},null]`. Suppose that we keep P_1. In that case, if $\Delta = 110$ and the local cost of joining `R` with any of the remaining query tables is over 10 units, we would get an infeasible solution because we cannot join P_1 without violating the cost threshold. Had we kept P_2 we could have obtained a solution. However, if we keep P_2 and Δ is higher, we could return a suboptimal solution that uses P_2 rather than the more efficient P_1.

The main problem is that the traditional principle of optimality does not hold in our scenario. That is, a subplan that is suboptimal in terms of deep cost might be part of the optimal execution plan due to having a smaller shallow cost. To correctly handle spool operators, we need to generalize the `Memo` data structure, so that it keeps all candidate plans that *might* become part of the optimal solution. Specifically, `Memo[R,S]` must contain, not just the plan P with the smallest value of $DC(P)$, but instead all plans in the two-dimensional *skyline* [1] of (*SC*, *DC*). Therefore, we extend the `Memo` data structure so that it returns a set of plans for each input `(R,S)` pair, and modify `updateMemo` to:

```
01  if (P ≠ null and ∀p ∈ P: SC(p)≤ Δ)
02     Memo[R, S] = skyline(Memo[R, S] ∪ P)
```

The last change we need to make to the algorithm in Figure 3 has to do with the search space itself. Since `Memo[`$\mathcal{R}$`,`$\mathcal{S}$`]` (and hence `optimize`) returns a *set* of plans rather than a single plan, we need to consider all different ways to combine such intermediate results into larger execution plans. For instance, lines 11 and 12 returns sets of plans in `CP1` and `CP2`. Therefore, we change lines 13-14 to:

```
13   foreach (pCP1, pCP2) ∈ CP1 × CP2
14     CP = JA(pCP1, pCP2)
```

and we make similar changes in lines 3-4. The resulting algorithm (denoted `optimize-S` in Figure 4) finds the optimal query slicing for a given threshold when using *Spool* operators.

3.2 Local Partitioned-Spools

A drawback of `optimize-S` is that it might fail to find *any* feasible solution for some values of Δ. Suppose, as a trivial example, that just scanning a base table already exceeds Δ. In this case, no matter where we place *Spool* operators, there would always be a subplan p for which $SC(p) > \Delta$, and thus `optimize-S` would not return any valid solution. In general, for a query q with k joins, it can be shown that `optimize-S` will not find a solution for $\Delta < cost(q)/(2k)$, where $cost(q)$ is the cost of the query obtained by calling `optimize(q, null)` (i.e., without Δ constraints).

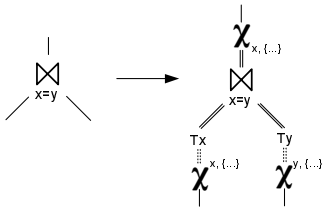


Figure 5: Local partitioned-spools.

To address the above shortcoming, we extend `optimize-S` to include *local* partitioned-spools. The idea is to also consider "surrounding" each operator with partitioned spools, and thus avoid having a single operator that is too big to fit in the threshold Δ. Figure 5 shows an example of partitioned spools surrounding a join. The join operator, which might be too large to fit the threshold Δ, is modified into a partitioned join, which would fit Δ by adjusting the column ranges appropriately. We next discuss the two main challenges to incorporate these alternatives into the search strategy, namely, how to instantiate a local partitioned spool with the proper column ranges, and how to enumerate the larger space of plans.

Obtaining column ranges. Suppose we are given a parametric plan like the one at the right of Figure 5, and we have to find the right ranges to instantiate in the *iSpool* and *oSpool* operators. Since the parametric plan contains *oSpool* operators right below the join, the cost of the sub-plans below such *oSpool* operators are independent of the actual ranges for the spool column (in that sense, the choice of column ranges is *local*). We can then leverage the cost model of the optimizer and search for partitions that minimize the overall execution cost. As in [12], we assume the fewer the partitions (and therefore the larger the work done per partition), the better the overall cost (however, see the discussion in Appendix B.2). Therefore, we always choose the largest possible ranges that result in a query slice instance that fits Δ. Figure 6 shows a simple procedure based on binary search that incrementally find ranges that make each *iSpool* iteration fit in Δ. Note that the actual technique to find ranges is orthogonal to the enumeration strategy itself, and thus we can replace the algorithm in Figure 6 by more sophisticated alternatives such as interpolation search or the *optimal-splitter* technique of [12].

Enumerating local partitioned spools. The original algorithm `optimize-S` considers in the search space all relevant plans with the template shown in Figure 5. Also, `optimize-S` considers putting a *Spool* operator on top of every plan it considers. Thus, given a join operator, it will consider execution plans for its children that are spooled at the top (those plans would be part of the `Memo` skyline, because the shallow cost of scanning a temporary table is minimal and cannot be dominated). Consider line 18 in Figure 4:

18 updateMemo($\mathcal{R}$, $\mathcal{S}$, Scan(Spool(CP)))

Plan `CP` is defined as `JA(pCP1, pCP2)` in line 16 for some join algorithm `JA` and subplans `pCP1` and `pCP2`. Whenever both `pCP1` and `pCP2` are themselves scans over temporary tables produced by a spool operator, the resulting plan `Scan(Spool(CP))` matches the template that we consider for local partitioned spools. We consider local partitioned plans by adding the following logic to `optimize-S`:

```
18.1  if (tempScan(pCP1) and tempScan(pCP2))
18.2     c = join column from pCP1
18.3     lpCP = Scan(iSpool_c(changeSpools(CP, c)))
18.4     findPartitions(lpCP)
18.5     updateMemo(R, S, lpCP)
```

Here, `tempScan(p)` determines whether `p` scans a temporary table produced by a spool operator. Therefore, in addition to regular *Spool* operators, the logic in lines 18.1-18.5 considers all possible local partitioned spools in the search space. It does so by picking every suitable plan pattern `CP`, and calling `changeSpools(CP, c)`, which replaces the top-most *Spool* or *iSpool* operator with *oSpool* or *ioSpool* operators in path from the root of `CP` to the leaf node that contains column `c` (modulo column equivalences). It then adds a new *iSpool* operator at the root, and calls `findPartitions` to instantiate a suitable partitioning strategy for the local partitioned spool. Thus, we can always find query slices for given values of Δ. We call the resulting algorithm `optimize-LPS`.

```
findPartitions (P:parametric plan)
returns RS:set of ranges
01 RS = ∅, L = −∞, H = ∞
02 while (L < H)
03    rMin = L, rMax = H
04    fMin = SC(P[L, rMin]), fMax = SC(P[L, rMax])
05    while (rMax-rMin > ϵ)
06       rMid = (rMin + rMax) / 2
07       fMid = SC(P[L, rMid])
08       if (fMid > Δ) rMax = rMid
09       else rMin = rMid
10    RS = RS ∪ {(L, rMid]
11    L = rMid
12 return RS
```

Figure 6: Finding ranges for partitioned spools.

Additional Details: We next discuss some details that we omitted earlier for simplicity. The first complication arises due to data skew. Suppose that value `R.x=10` in table `R` is repeated so many times that the local cost of performing a partitioned join `R` $\bowtie_{x=y}$ `S` with value `R.x` $=$ `S.y` $= 10$ already exceeds Δ. Since we cannot further subdivide `R.x=10`, `optimize-LPS` would return no solution. Similar to techniques used in parallel database systems, we can extend the partitioning algorithm so that it also considers secondary partitioning columns in case of extreme skew. For instance, we can subdivide `R.x=10` into `R.x=10 and R.id` $\in \{(-\infty, 100], (100, \infty]\}$, where `R.id` is another column in `R` (preferably a key). Each secondary partition of `R.x=10` has to join with the partition `S.y=10` in `S`. If both `R` and `S` are subdivided for the same value, the cross product of joins is performed. The second detail in lines 18.1-18.5 above is that it assumes a single join predicate between the `lpCP1` and `lpCP2`. In general, if the join graph contains cycles or the search space includes bushy trees, there might be more than a single join predicate. In such a case, we execute lines 18.2-18.5 for each join predicate. Finally, a subtle detail is related to the cost model for *oSpool* operators. The cost of an *oSpool* operator is not the same as that of a regular *Spool* operator. An *oSpool* operator needs to additionally evaluate range predicates to determine the temporary table over which the current input tuple should be appended. The number of range predicates depends on the number of temporary tables, but this number is not known in advance, as it is only determined after calling `findPartitions`. Algorithm `optimize-LPS`

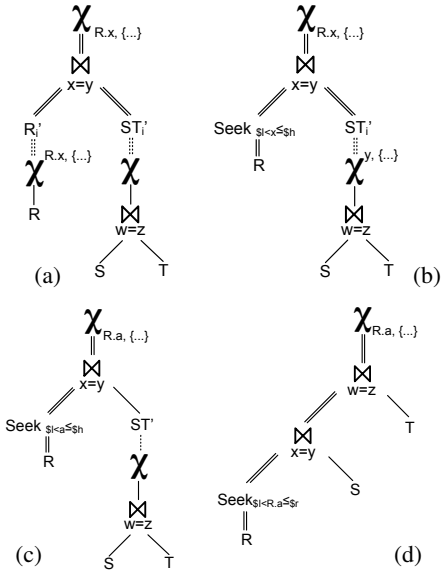


Figure 7: General partitioned spools.

assumes that a single partition would be required when constructing the skyline bottom-up, and then modifies *Spool* operators into the required *oSpool* operators in lines 18.3 using `changeSpools`. A corner case happens when the optimal `pCP1` (respectively `pCP2`) (barely) fits Δ, but the modified subplan which uses *oSpool* does not when adding the required range predicates. This would result in missing a valid alternative plan `pCP1'` that, while dominated by `pCP1`, has the possibility to perform the required range predicates within Δ. To address this limitation, we relax the dominance condition in the skyline computation of `updateMemo`, so that whenever p_1 dominates p_2, both p_1 and p_2 are scans over temporary tables, and p_2's shallow cost of its spool child is smaller than that of p_1, we do not prune p_2 from the skyline.

3.3 General Partitioned Spools

Although local partitioned spools always return feasible solutions, there are scenarios (e.g., when leveraging existing indexes) for which `optimize-LPS` returns suboptimal plans. Consider the local partitioned spool for a three-way join on tables `R`, `S` and `T` (see Figure 7(a)). Assume that a *covering* index on `R.x` is available (i.e., an index that contains all required columns from `R`). We can replace the $oSpool^{(x,\{\ldots\})}$ operator by an access path that directly retrieves the tuples in `R` satisfying each range predicate over `R.x` (see Figure 7(b)). If the remaining single-table predicates on `R` are not very selective, this alternative can be more efficient than materializing intermediate results. Now suppose that `R` is originally accessed in Figure 7(a) using an index over column `R.a` (say there is a single-table predicate on such column). An alternative similar to the plan in Figure 7(b) is shown in Figure 7(c). This plan partitions table `R`, not on the join column `R.x`, but instead on `R.a` (and thus it is not a partitioned join). Then, for each range in `R.a`, the join is performed with the whole right-side relation (which in the figure is spooled into temporary table `T`). Figure 7(d) shows another alternative that uses a *deep partitioning* of column `R.a`. Each partition of `R` is joined with both `S` and `T` before the partial result is written into the common temporary table.

Depending on cardinality values and index availability, each alternative in Figure 7 might be optimal. The plans in Figure 7(b-d), however, are not found by `optimize-LPS`, since there is no *oSpool* that immediately closes the top-most *iSpool* operator.

We next discuss how to extend `optimize-LPS` to exploit arbitrary placement of all spool variants. Since `optimize-LPS` places spools tightly surrounding join operators, there is no need to handle parametric selection predicates on execution subplans (the implicit parametrization is done locally by the corresponding *oSpool* operators and the choice of column ranges is local). When considering the full space of plans, however, we need to explicitly create and propagate parametric plans. Function `paramCols` in Figure 11 returns the set of columns that a given plan is parameterized upon. For plans that do not have a spool operator at the root, `paramCols` always returns a single column (since we consider single-column partitioned spools), or `null` if the plan is not parameterized. In contrast, if the plan p does have a spool operator at the root, the set of parameterized columns are all those in join predicates between a table in p and a table not in p. These columns would eventually be used by `changeSpools` to instantiate *oSpool* operators.

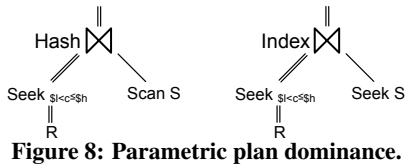


Figure 8: Parametric plan dominance.

A distinguishing feature of parametric plans is that we do not know their costs until we instantiate the parameters. For that reason, we cannot prune away a parametric plan p unless we are sure that p will be dominated by other plans for *all possible range instances*. Figure 8 shows a simple example where a hash-join and an index-join alternatives might dominate each other depending on the number of tuples satisfying the predicate on the outer table `R`. Specifically, the dominance condition on the skyline operator needs to be extended so that (i) plans parameterized on different columns do not dominate each other, and (ii) parametric plan p_1 dominates parametric plan p_2 (parameterized on the same column) whenever p_1 dominates p_2 for every parameter instance.

The main algorithm for dealing with arbitrary spool variants, which we call `optimize-PS` is discussed in detail in Appendix A.1. Specifically, we show how to generalize the dominance condition on the skyline operator, how to generate parametric plans for interesting columns, and how to generate join combinations.

3.4 LPS with Single Table Optimization

The generic algorithm `optimize-PS` discussed above traverses the full space of extended execution plans and considers all spool variants. However, due to the large number of parametric plans that are generated (and thus generally not pruned), `optimize-PS` is usually much more expensive than the restricted variants discussed in Sections 3.1 and 3.2. At the same time, resulting plans by `optimize-PS` are of better quality because of the extended search space that is considered. We next introduce `optimize-LPS*`, a technique that generalizes `optimize-LPS` (and uses slightly more resources), but gives results closer to those of `optimize-PS`.

As motivation, consider again the examples in Figure 7(b-c). A common property of these extended plans is that whenever an *iSpool* is placed on top of an operator p, either it is closed by an *oSpool* operator immediately below p, or else the partitioning column (modulo join equivalences) is defined over a *single-table* subplan of p. This is important because parametric plans are therefore only defined for single-table expressions, and therefore do not propagate arbitrarily upwards in the enumeration strategy. Since we can check for dominance of such parametric plans easily, complex skyline computations (or heuristic approximations) are not needed.

Figure 7(d) shows a plan that does not fall in the category explained above, because it uses a *deep* partitioning of column R.a. However, note that such a plan necessarily executes multiple joins between partitions of R and the whole of S and T. If the joins are hash- or merge-based, S and T would be read multiple times. If the joins are index-based, it means that some intermediate result is small, and we could materialize such result earlier with a relatively small penalty. Therefore, the plan in Figure 7(d) requires rather specific circumstances to be significantly better than alternatives. This analysis motivates optimize-LPS*, which extends optimize-LPS by allowing single-table parametric plans that can take advantage of index strategies. We can obtain optimize-LPS* by restricting the classes of joins that we consider in optimize-PS, as shown in Appendix A.2. In general, optimize-LPS* produces plans that are comparable to those given by optimize-PS at a fraction of optimization time.

3.5 PS with Plan Pattern Optimization

We previously explained how optimize-LPS* reduces the overhead of optimize-PS by restricting the places on which spool operators can be located (e.g., we forbid deep partitioned columns). In this section we explore an alternative approach, in which we restrict the plans on which spool operators can be placed (without restricting spool placement on such plans whatsoever). Our technique, which we call optimize-PS*, can be seen as a generalization of the post-processing techniques in parallel databases that "sprinkle" parallelism over the best serial plan. Specifically, optimize-PS* considers spool operators over plans that share the same *pattern* with the optimal plan found without Δ constraints. Two plans share the same pattern if the join tree is the same modulo commutativity (join algorithms can change, though).

Therefore, optimize-PS* starts by calling optimize (see Figure 3) and obtaining the optimal plan $\mathtt{P}_{opt}$ independent of Δ. Then, it proceeds very similarly to optimize-PS, but only exploring the relevant plan fragments that appear in the optimal plan. The simple extensions required to implement optimize-PS* are discussed in Appendix A.3. Algorithm optimize-PS* is much faster than optimize-PS because it only considers a small number of execution plans. It might miss opportunities, however, since slicing the optimal plan is not the same as obtaining the optimal query slicing.

3.6 Summary of Techniques

Table 9 summarizes both the search space enumerated by the techniques (in order of generality), and also the distinguishing features involved in their solutions. These strategies balance optimization time with the quality of resulting extended execution plans. Note that throughout this section we focused on SPJ queries to simplify the presentation. Appendix B discusses several important extensions and optimizations, such as handling GROUP BY) and other operators, more details on partitioning strategies, and various performance improvements.

	S	LPS	LPS*	PS / PS*
Space	χ	+local χ^{Co}_{ci}	+single tables	Full / Optimal
Features	(*SC*,*DC*) skyline	+binary search	+single-table parametric plans	+cost skyline and parametric plans

Figure 9: Summary of optimization strategies.

4. EXPERIMENTAL EVALUATION

In this section we report an experimental evaluation of the techniques described in this paper. We implemented the different query slicing algorithms of Section 3 by extending the exhaustive optimizer in [2]. The optimizer cost model was ported from that of Microsoft SQL Server's optimizer. Unless explicitly stated otherwise, we use binary search for determining range partitions, and an early search bailout of 0.1% (see Appendix B.3). We used the workload generator discussed in [2] to produce a synthetic queries, which allowed us to vary different factors like the number of tables and their sizes, join topologies, predicate selectivities and availability of indexes. Query templates follow chain, snowflake, and star schemas with foreign-key joins, optionally include single-table local selection predicates (with random selectivity in the range 0.1%-10%) and group-by clauses. Table sizes range from kilobytes to gigabytes. For the case of snowflake schemas, workloads look similar to those in a typical 10GB TPC-H database.

4.1 An Illustrative Example

To illustrate the different plans considered by our techniques, we took a four-way star-join query and explored how the overall cost of the query varies with decreasing values of Δ using optimize-PS (see Figure 10). When $\Delta = \infty$ the overall cost is 12.5 units. As we decrease Δ, the overall execution time gradually increases up to 25 units for $\Delta = 0.2$. The figure also shows selected extended execution plans for certain values of Δ. The query result size is rather small, so when Δ is slightly below the cost of the optimal plan, the best extended plan in Figure 10(a) puts a *Spool* operator at the root. For $\Delta = 11.1$ intermediate results become too expensive, so a second *Spool* operator is placed on top of the first join in Figure 10(b). For even smaller $\Delta = 7.1$, there is no plan that exclusively uses *Spool* operators, and the optimal plan in Figure 10(c) introduces a top-most *iSpool* with a deep partitioning attribute on table $\mathtt{T}_0$. The materialized table $\mathtt{T}_{23}$ is read multiple times, once per partition on $\mathtt{T}_0$.c. When we further decrease Δ down to 1.9 units, the *Spool* operator on top of tables $\mathtt{T}_2$ and $\mathtt{T}_3$ is transformed into a second *iSpool* operator that induces a partitioned join. However, $\mathtt{T}_2$ cannot be read completely under Δ and therefore a third *ioSpool* operator, which repartitions $\mathtt{T}_2$, is introduced. The cost of the optimal plan gracefully degrades for smaller values of Δ, and the resulting plans leverage all variants of *Spool* operators.

4.2 Summary of Experimental Results

We next summarize our experimental results, and refer to Appendix C for quantitative information that supports our findings.

Optimizer Efficiency: In our experiments, optimize-PS becomes prohibitively expensive for queries with around or over 8 joins. All other alternatives are practical for the whole range of workloads, taking less than 400 msec. on average to optimize the most expensive 10-way star-join workload. Also note that optimize-LPS* is cheaper than optimize-PS* for chain queries, but the trend reverses for more complex join topologies, and for star queries with 8 or more tables, optimize-PS* is the cheapest alternative overall.

Plan Quality: For each query in the workload and Δ threshold, we define the overhead ratio as the optimizer cost of the optimal extended execution plan of our techniques divided by the optimizer cost of the optimal execution plan with no Δ threshold. An overhead ratio of 1.25 for $\Delta = C/4$ means that the optimal query slicing P_S is 25% worse than the optimal –unsliced– plan P_U, when no slice in P_S is allowed to use more than 25% of the overall cost of P_U. optimize-S does not produce a plan for the vast majority of cases. optimize-LPS is the simplest technique that results in valid queries for arbitrary Δ values. However, the overhead ratios are significantly higher than those of the more advanced strategies. Finally, optimize-LPS* and optimize-PS* are almost identical in quality to the optimal optimize-PS (under 2% difference).

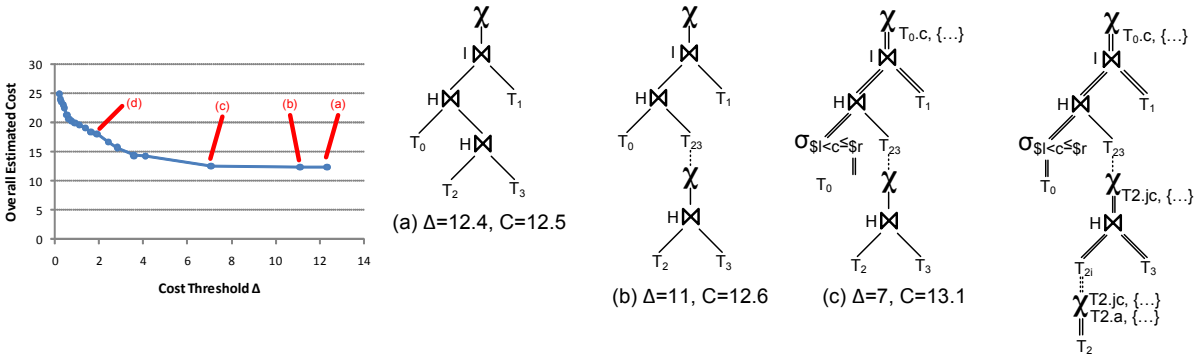


Figure 10: Optimizing a 4-way join for varying Δ thresholds.

5. RELATED WORK

Managing long-running queries is an important problem in data warehousing. A study of current workload management policies is presented in [9, 10]. Techniques can be classified into admission control, scheduling, and execution control. While most systems use combinations of these techniques to manage long-running queries, designing a truly robust technique remains an open research problem. Our query slicing techniques are applicable to various aspects of resource management by slicing complex queries into pieces that respect a cost threshold Δ. There has been recent work on new server mechanisms to pause and resume a long-running query (e.g., [3, 4]). These techniques are an interesting addition to the repertoire of execution control mechanisms. In general, admission control techniques need to be used in conjunction with execution control mechanisms and it is interesting to examine how to best combine query slicing techniques proposed in this paper with appropriate execution control techniques (e.g., Pause/Resume).

The partitioned spool operator used in this paper is similar to the "split" operator used in parallel database systems [6]. The split operator partitions its output stream (using a split table) to an appropriate process while the *oSpool* operator partitions its output stream to temporary tables. While the problem of choosing an appropriate partitioning of an intermediate result in a query tree has been previously studied in the context of parallel query optimization [7, 8], there are a number of differences. First, we need to handle the additional constraint of a cost threshold, which significantly impacts the resulting techniques. Second, physical design plays an important role in our search space. Typically, in parallel query optimization, the set of columns that are "interesting" for partitioning are usually the columns on which the join predicates are defined. In contrast, a column on which there is a covering index for a relation could still serve as an interesting partitioning column (see Figure 7(b)) because it can potentially lead to a plan in which all the slices respect the cost constraint with no materialization. Finally, some techniques in parallel databases exploit the current layout of data (e.g., using small tables that are replicated in all nodes for join processing). However, these techniques do not consider whether to replicate a table during optimization. The search space of our techniques include and generalize the equivalent of these strategies by placing spool operators over small intermediate results.

6. CONCLUSIONS

In this paper we introduce the idea of *query slicing*, or dividing a complex long-running query into components that are estimated to run in a predefined amount of time. We studied a spectrum of techniques for query slicing that extend the traditional optimization search space with different tradeoffs between optimization time and the quality of the "sliced" plan. Our experimental results indicate that `optimize-LPS*` and `optimize-PS*` are almost undistinguishable in terms of quality and result in the best tradeoff between optimization runtime and quality of resulting extended execution plans.

7. REFERENCES

[1] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.

[2] N. Bruno, C. Galindo-Legaria, and M. Joshi. Polynomial heuristics for query optimization. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2010.

[3] B. Chandramouli, C. Bond, S. Babu, and J. Yang. Query suspend and resume. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2007.

[4] S. Chaudhuri et al. Stop-and-restart style execution for long running decision support queries. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2007.

[5] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. In *Proceedings of the 16th International Conference on Data Engineering*, 2000.

[6] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *In Communications of the ACM*, 35(6), 1992.

[7] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1992.

[8] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1995.

[9] S. Krompass, U. Dayal, H. A. Kuno, and A. Kemper. Dynamic workload management for very large data warehouses: Juggling feathers and bowling balls. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2007.

[10] S. Krompass et al. Managing long-running queries. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2009.

[11] Microsoft Corporation. SQL Server 2008 Books Online. Accessible at `http://msdn.microsoft.com/en-us/library/ms190419.aspx`.

[12] K. A. Ross and J. Cieslewicz. Optimal splitters for database partitioning with size bounds. In *Proceedings of the International Conference on Database Theory*, 2009.

[13] P. G. Selinger et al. Access path selection in a relational database management system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1979.

[14] C. D. Weissman and S. Bobrowski. The design of the Force.com MultiTenant Internet Application Development Platform. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2009.

[15] W. Yan and P. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1995.

[16] C. Yang et al. Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2010.

APPENDIX

A. ALGORITHMIC DETAILS

A.1 General Partitioned Spools

In this section we discuss details for optimize-PS, which can deal with arbitrary spool variants. Function updateMemo in Figure 11 generalizes that of optimize-LPS in two aspects. First, line 1 checks $SC(p) \leq \Delta$ only for non-parametric plans. Second, the dominance condition on the skyline operator is extended so that (i) plans parameterized on different columns do not dominate each other, and (ii) parametric plan p_1 dominates parametric plan p_2 (parameterized on the same column) whenever p_1 dominates p_2 for every parameter instance[4]. This condition can be very difficult to test and in general involves detailed knowledge of the cost model. A heuristic that works very well in practice is to try extreme selectivity ranges (say ϵ and $1-\epsilon$) for the parametric predicate, and declare that p_1 dominates p_2 if it does it for both data points (similar to the MNSA technique in [5]). This is correct when cost lines of both plans do not intersect more than once, and a heuristic otherwise.

We now discuss the main algorithm for dealing with arbitrary spool variants, which we call optimize-PS in Figure 11. The first difference with respect to optimize-LPS is on lines 6-13, which generate single-table execution plans. In addition to plans obtained by previous techniques, lines 10-13 generate parametric plans for every *interesting* column. A column is interesting if it is either part of a join predicate in the query, or it is a key column of an index. There could be more than a single plan for a given column, to cover the whole range of selectivity values. Consider a subquery $\sigma_{R.a<10}(R)$ and column $R.b$. Line 11 would generate a plan that seeks I_b for $1≤b<$h, fetches the remaining columns and then applies R.a<10 on the fly (for low selectivity ranges on b). Additionally, it will generate a plan that uses an index on R.a to obtain the tuples that satisfy R.a<10 and then apply the range predicate on R.b on the fly. If the query processor handles index intersection plans, additional plans might be generated in line 11. All such parametric plans are stored in Memo[$\mathcal{R}$,$\mathcal{S}$], as any of them could be part of the overall optimal plan.

The second difference is how joins are generated in lines 14-20. Rather than just considering plain spools and the extensions of optimize-LPS for local partitioned spools, optimize-PS calls function generateJoins for each combination of plans pCP1 and pCP2 and join algorithm JA (contrast generateJoins with lines 16-18.5 in optimize-LPS). Function generateJoins considers each combination of parametric column for input plans P1 and P2 (recall that except for plans with root spools, each plan has a single parametric column or is null). If at most one of P1 and P2 has a non-null parametric column, or both have parametric columns that are joined together in P1 $\bowtie$ P2, we can generate a new plan that is either parametric on one column or not parametric at all (depending whether either P_1 or P_2 are parametric to begin with). In that case, lines 3-4 generate the –potentially parametric– plan CP, and lines 5-10 the corresponding plan that uses spool variants.

A.2 LPS with Single Table Optimization

As discussed earlier, optimize-LPS* extends optimize-LPS by allowing single-table parametric plans that can take advantage of index strategies. We obtain optimize-LPS* by simply restricting the considered join classes in generateJoins in Figure 11:

```
1.1   if (c1≠null and ¬validForLPS*(P1)) continue
1.2   if (c2≠null and ¬validForLPS*(P2)) continue
```

[4]Strictly speaking, p_2 is dominated whenever there is a plan (not necessarily the same) in the skyline that dominates p_2 for every parameter instance.

```
paramCols (P:plan)
returns columns for which P is parameterized
01 C = parametric(P) ? {parameter(P)} :  {null}
02 if (tempScan(P))
03    C = C ∪ {c in cols(P):(c=c') is join predicate}
04 return C
```

```
updateMemo (R:tables, S:order, P:plan)
01 if (P ≠ null and ∀p∈P:¬parametric(P)⇒SC(p)≤ Δ)
02    Memo[R, S] = skyline(Memo[R, S] ∪ P)
```

```
generateJoins (P1,P2:plan, JA:join algorithm)
01 for each (c1,c2) in paramCols(P1) × paramCols(P2)
02    if (c1=null or c2=null or (c1=c2) are joined)
03       c = (c1=null) ?  c2 :  c1
04       CP = JA(pCP1, pCP2)
05       updateMemo(R, S, CP)
06       if (c=null)
07          pCP = Scan(Spool(CP))
08       else
09          pCP = Scan(iSpool_c(changeSpools(CP, c)))
10          findPartitions(pCP)
11       updateMemo(R, S, pCP)
```

```
optimize-PS (R:tables, S:order)
returns skyline of plans for R satisfying S
01 if (Memo[R, S] was not yet calculated)
02    if (S ≠ null)
03       for each (CP ∈ optimize-PS(R, null))
04          updateMemo(R, S, Sort(CP))
05          updateMemo(R, S, Scan(Spool(Sort_S(CP))))
06    if (|R| = 1)
07       CP = best plan under S order
08       updateMemo(R, S, CP)
09       updateMemo(R, S, Scan(Spool(CP)))
10       for each 'interesting' column C // see Section 3.3
11          CPS = parametric plans for σ_{$l≤C<$h}(R)
                  using IC under S order
12          for each CP in CPS
13            updateMemo(R, S, CP)
14    else for each valid partition (R1, R2) of R
15       for each join algorithm JA
16          S1,S2 = required orders of R1,R2 for JA
17          CP1 = optimize-PS(R1, S1)
18          CP2 = optimize-PS(R2, S2)
19          for each (pCP1, pCP2) ∈ CP1 × CP2
20            generateJoins(pCP1, pCP2, JA)
21 return Memo[R, S]
```

Figure 11: Handling all *Spool* variants for query slicing.

where validForLPS* accepts plans that either have a spool operator at the root, are defined over a single table, or else are not parametric. That is, validForLPS*(p) is equivalent to:

```
tempScan(p) ∨ singleTable(p) ∨ ¬parametric(p)
```

A.3 PS with Plan Pattern Optimization

Algorithm optimize-PS* is similar to optimize-PS, but only explores plan fragments that appear in the optimal plan. For that purpose, we need to slightly modify the search strategy in algorithm optimize-PS, which originally iterates over all possible partitions of the input tables, so that only trees that share their patterns with the optimal P_{opt} are explored. Specifically, we need to change line 14 in optimize-PS to:

```
14    else for each (R1, R2) sharing P_opt's pattern
```

B. EXTENSIONS AND OPTIMIZATIONS

B.1 Handling Additional Operators

Section 3 focuses exclusively on SPJ queries, but the techniques discussed there can be analogously extended to other operators. Non-partitioned spool operators χ are straightforward to manipulate with arbitrary operators since they can be placed anywhere in a tree. Partitioned spool operators, however, require additional care and the details are dependant on the operators they operate upon.

Consider the Sort operator, which orders an input relation by a given column. If we want to slice $\texttt{Sort}_a\texttt{(R)}$ we can locally partition the sort operator by using:

$$\chi_{(a,\{\ldots\})}\texttt{(Sort}_a\texttt{(Scan(}\chi^{(a,\{\ldots\})}\texttt{(R))))}$$

for suitable ranges over column a. If R can deliver ordered subranges of a by using an index plan, extensions to optimize-LPS* can also consider such strategies. The only requirement is that each iteration of the $\chi_{(a,\{\ldots\})}$ operator be done in increasing order of a ranges, so that the temporary table is fully sorted by a.

Consider now a group-by clause $\texttt{GB}_{\texttt{a,count(*)}}\texttt{(R)}$ with grouping column a. A local partitioned spool strategy can simply partition the input relation R by column a exactly as for the Sort case above. Since all tuples with the same a value belong to the same range, this strategy works with no further changes. If R can be partitioned by other columns due to some access path strategy, we can apply the techniques in [15] and decompose the group-by into local and global parts. For instance, if R can be partitioned by column c due to an index strategy, we can partition $\texttt{GB}_{a,count(*)}\texttt{(R)}$ as follows:

$$\texttt{GB}_{a,sum(B)}\ \left(\chi_{(c,\{\ldots\})}\texttt{(GB}_{a,B=count(*)}\texttt{(}\sigma_{\$l<c\leq\$h}\texttt{(R)))}\right)$$

Other operators can be handled analogously.

B.2 Revisiting Partitioning Strategies

We next present extensions to the basic partitioning strategies for query slicing and the challenges they introduce.

Optimal Partitioning for Opaque Cost Functions: Our partitioning techniques choose the largest possible ranges that result in a query slice instance that fits Δ. This technique is optimal if the cost model satisfies that $cost(P) \leq cost(P_1) + cost(P_2)$ where P_1 and P_2 partition P. In general, however, complex cost functions might not satisfy this property. Consider $R \bowtie S$ using a hash join alternative. If the whole join spills to disk but we use a 2-way partitioned join for which both iterations do not spill to disk, the partitioned strategy can be cheaper than the full join. This behavior is also applicable for regular optimization, but current optimizers do not handle such scenarios. Obtaining the optimal partitioning to minimize global cost is an open problem, and our techniques approximate that goal by maximizing the work per partition.

Multi-dimensional Partitioning: Our techniques handle a single partitioning column in a χ operator. In general, we can partition spools by multiple columns simultaneously, as long as the sub-plans iterate over the cross product of the column ranges (note that partitioned joins do not strictly partition over two different columns because these are joined together and thus can be treated as one). While multidimensional partitioning can result in better plans for some scenarios, the corresponding techniques are considerably more difficult and expensive. In this paper we take an approach that is analogous to avoiding cross-products in traditional optimization and do not consider multiple partitioning columns on spool variants.

Heterogeneous plans: The techniques in this paper enforce that every iteration of an *iSpool* operator executes the same plan. In some situations, it might be more efficient to execute different plans depending on the actual range predicates (e.g., for subplans of a partitioned join with skewed distribution in the join columns). As with multidimensional partitioning, this optimization might bring some improvement in execution time in some scenarios but it would significantly increase optimization time.

B.3 Performance Improvements

The main overhead of our techniques with respect to the traditional dynamic programming strategy comes from (i) considering many additional plans due to less aggressive pruning by the (*SC*,*DC*) cost skyline, and (ii) significant number of cost evaluations when searching for ranges. We next discuss heuristics that reduce such overhead at the expense of slight quality degradation.

Early binary-search bailout: The binary-search technique to identify partition ranges discussed in Section 3.2 can incur significant overhead at the last iterations, where considering a handful of different values does not change costs that much (and in any case, cost estimation is done using histograms, which are approximations of the real data). An alternative approach is to stop the binary search whenever successive attempts result in a selectivity change of, say, 1%. This way, we would still return a valid range that fits Δ, but the range would not be as tight as possible. At the same time, optimization time is reduced significantly, since the binary-search module is in the inner loop of our various techniques.

Skyline reduction: A heuristic to reduce the number of plans considered during optimization is to prune the plan skylines in the Memo structure (we suggest a similar approach with respect to extreme selectivity ranges in Section 3.3). Rather than keeping all plans that are not dominated in term of (*SC*,*DC*) costs, we can simplify the dominance test and keep plans that are not dominated either in *SC* or *DC* costs. In other words, we only keep two plans in each Memo cell, the one with the smallest *SC* cost, and the one with the smallest *DC* cost. This way, we reduce the number of plans that are considered and thus make optimization faster, at the expense of missing some alternatives.

B.4 Reoptimization/Cardinality Estimation

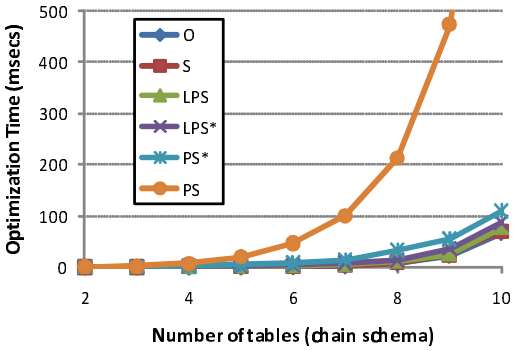
By design, our techniques depend on cost estimates produced by the optimizer, and indirectly on cardinality estimates. These estimates can sometimes be inaccurate, and there have been different approaches in the literature to improve cardinality estimation and optimization in general. These techniques are orthogonal to our approach and can certainly be leveraged. An interesting approach is to combine query slicing with ideas on incremental execution. As slices are executed, we can gather information about the materialized slices and, if they deviate from the original estimates, we can re-optimize the remaining query fragment. It is interesting to note that the local partitioned-spools of optimize-LPS and optimize-LPS* might be a better alternative than the potentially deep partitioning strategies of optimize-PS and optimize-PS*, as they would catch and reduce cardinality errors sooner.

C. EXPERIMENTAL RESULTS

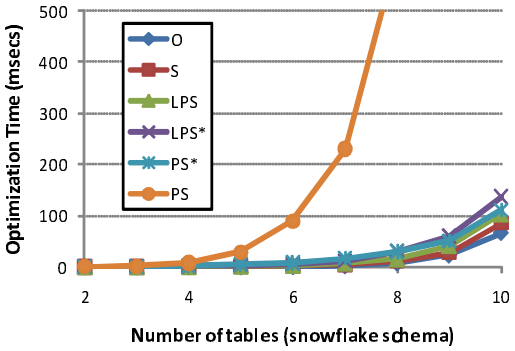
In this section we provide experimental results that support our conclusions in Section 4.

C.1 Optimization Efficiency

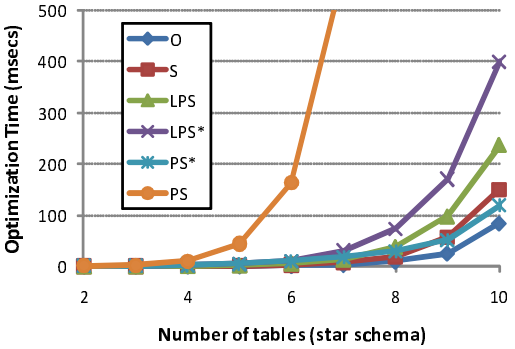
We first measure the efficiency of the different techniques discussed in Section 3 and also optimize-0, which is the traditional dynamic programming algorithm of Figure 3, which does not consider Δ thresholds. We generated 100-query random workloads using different join topologies and covering indexes for a large fraction of query predicates. We then optimized each query using a



(a) Chain topology.

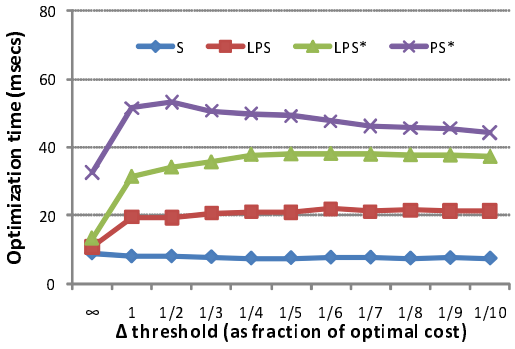


(b) Snowflake topology.

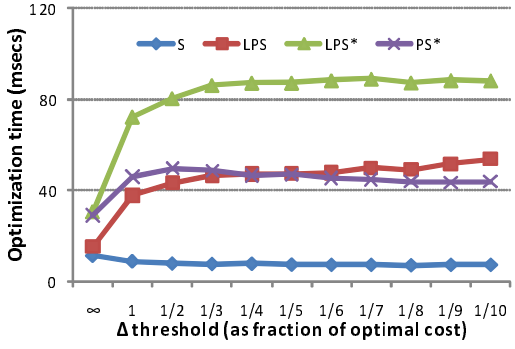


(c) Star topology.

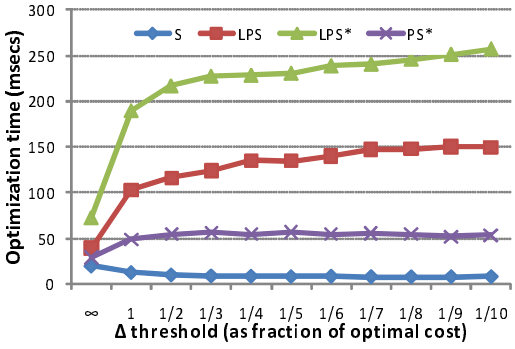
Figure 12: Elapsed time of different query slicing techniques for varying query complexity.



(a) Chain topology.



(b) Snowflake topology.



(c) Star topology.

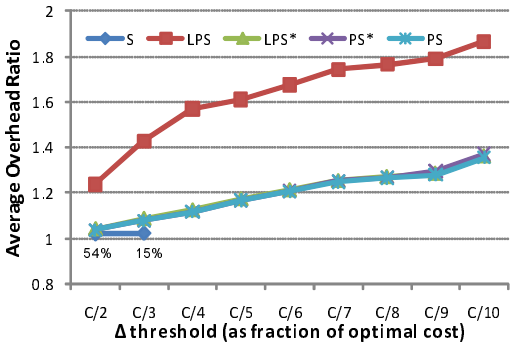
Figure 13: Elapsed time of different query slicing techniques for varying Δ values.

value of $\Delta = \infty$ and measured the elapsed time of each optimization technique and also the plain dynamic programming alternative (which does not perform query slicing). Figure 12 shows the average elapsed time of each algorithm for various 100-query workloads varying the number of joins.

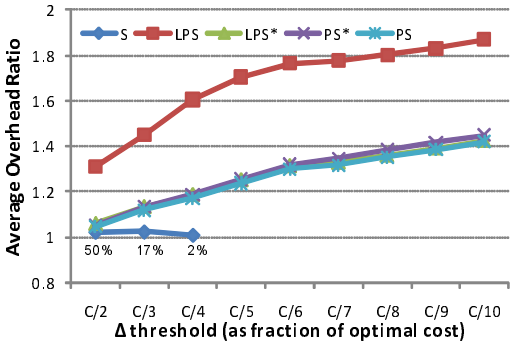
The figure shows that the join topology has some influence in the relative performance of the techniques. Specifically, chain queries are cheaper to process than star queries, and snowflake queries lie in between. In all experiments, `optimize-PS` quickly becomes prohibitively expensive for queries with around or over 8 joins. All other alternatives are practical for the whole range of workloads, taking less than 400 msec. on average to optimize the most expensive 10-way star-join workload. Also note that `optimize-LPS*` is cheaper than `optimize-PS*` for chain queries, but the trend reverses for more complex join topologies, and for star queries with 8 or more tables, `optimize-PS*` is the cheapest alternative overall among those that consistently generate valid solutions. Note that `optimize-S` (and for that matter, the baseline dynamic programming), though faster than `optimize-PS*`, very frequently fails to find any solution.

We noticed that with decreasing values of Δ, the optimization time of our techniques initially increases, but at some inflection point the trend is reversed. In qualitative terms, this can be explained as follows. For very large values of Δ, partitioning techniques are usually not required, because the whole input can be spooled completely (if necessary), and thus the number of plans in the `Memo` skylines is smaller. As the value of Δ decreases, the techniques increase in optimization time due to larger skylines and longer range searches. At some point, however, Δ becomes small enough that many plans are pruned from the `Memo` because they do not fit in Δ, and the overall cost of the techniques starts slowly decreasing. This inflexion point, however, is different for each technique and also depends on query characteristics.

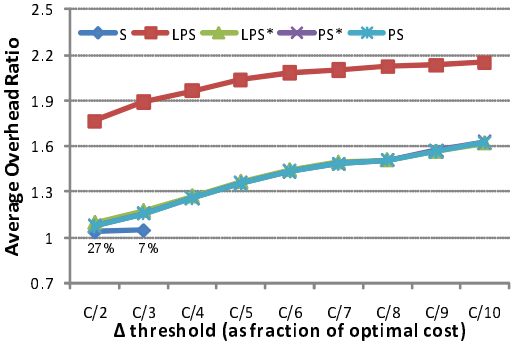
Figure 13 shows how optimization time varies when changing Δ values for 8-way join queries (we omit `optimize-PS` since it is



(a) Chain topology.



(b) Snowflake topology.



(c) Star topology.

Figure 14: Quality of different query slicing techniques for varying query complexity.

significantly more expensive than the alternatives). Specifically, we consider Δ=∞ and also vary Δ from C to $C/10$ for each query, where C is the cost of the optimal plan by `optimize-O` (i.e., without Δ thresholds). Again, the join topology influences the trends. For chain queries, the inflexion point is reached sooner for all techniques, and `optimize-PS*` is the least efficient alternative. This trend is reversed for star queries, where `optimize-PS*` is the most efficient technique among those that use partitioned spools. The reason is that `optimize-PS*` is intrinsically more expensive, but only operates over a single tree, and therefore benefits from complex optimization instances. Figure 13(b) clearly shows that the inflexion point of `optimize-PS*` is smaller than that of `optimize-LPS`, and the optimization costs cross each other at around $C/3$.

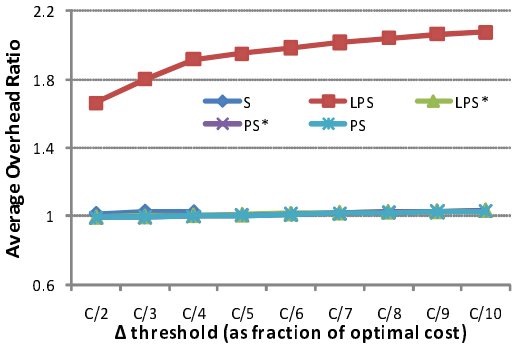


Figure 15: Quality for chain queries with no indexes.

C.2 Overall Plan Quality

We now examine the quality of the plans returned by different optimization strategies. For that purpose, we consider again the 100-query workloads with 8-way joins of the previous section (the results extend for other query sizes). For each query in the workload and given Δ threshold, we define the overhead ratio as the optimizer cost of the optimal extended execution plan of our techniques divided by the optimizer cost of the optimal execution plan obtained by `optimize-O` (i.e., with no Δ threshold). For instance, an overhead ratio of 1.25 for $\Delta = C/4$ means that the optimal query slicing P_S is 25% worse than the optimal –unsliced– plan P_U, when no slice in P_S is allowed to use more than 25% of the overall cost of P_U.

Figure 14 shows the results for different join topologies. We make the following observations. First, `optimize-S` does not produce a plan for the vast majority of cases. For instance, in Figure 14(a), only 54% of queries have answers for Δ=$C/2$, just 15% for Δ=$C/3$, and none for smaller values of Δ. Since overhead ratio averages are computed only for successful optimizations, it seems that `optimize-S` performs better overall than the other techniques, when in reality is just the opposite. In fact, `optimize-LPS` is the simplest technique that results in valid queries for arbitrary Δ values. However, the overhead ratios are significantly higher than those of the more advanced strategies. Finally, we see that `optimize-LPS*` and `optimize-PS*` are almost indistinguishable in terms of quality from the optimal `optimize-PS`. In general, the best overall is `optimize-PS`, but the difference is below 2% in all cases.

We comment on an interesting result that we obtained when we repeated the experiment of Figure 14(a) but changing the physical design so that only clustered indexes were available. Figure 15 shows overhead ratios that at first sight suggest virtually no overhead for optimizing such queries even for values of $\Delta = C/10$. Upon closer inspection, we found that more than half of the queries behave similarly to the case in Figure 14(a), but a significant fraction of the remaining queries result in overhead ratios smaller than one. This happens because in absence of indexes, most joins are hash-based. In such cases, evaluating a partitioned hash-join requires materializing join results (which can be small) but avoids spilling join inputs to disk (which can be large). Therefore, plans that use partitioned spools can be more efficient than the optimal unsliced plans. This strategy, although not implemented in practice, is also applicable for regular query optimization.