Query Suspend and Resume*

Badrish Chandramouli
Duke University
badrish@cs.duke.edu

Christopher N. Bond†
Google Inc.
chrisbond@google.com

Shivnath Babu
Duke University
shivnath@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

Suppose a long-running analytical query is executing on a database server and has been allocated a large amount of physical memory. A high-priority task comes in and we need to run it immediately with all available resources. We have several choices. We could swap out the old query to disk, but writing out a large execution state may take too much time. Another option is to terminate the old query and restart it after the new task completes, but we would waste all the work already performed by the old query. Yet another alternative is to periodically checkpoint the query during execution, but traditional synchronous checkpointing carries high overhead. In this paper, we advocate a database-centric approach to implementing query suspension and resumption, with negligible execution overhead, bounded suspension cost, and efficient resumption. The basic idea is to let each physical query operator perform lightweight checkpointing according to its own semantics, and coordinate asynchronous checkpoints among operators through a novel contracting mechanism. At the time of suspension, we find an optimized suspend plan for the query, which may involve a combination of dumping current state to disk and going back to previous checkpoints. The plan seeks to minimize the suspend/resume overhead while observing the constraint on suspension time. Our approach requires only small changes to the iterator interface, which we have implemented in the PREDATOR database system. Experiments with our implementation demonstrate significant advantages of our approach over traditional alternatives.

Categories and Subject Descriptors: H.2.4 [Database Management]: Query Processing

General Terms: Design, Experimentation, Performance

Keywords: Query, Suspend, Resume, Processing, Optimization

1 Introduction

Consider a database management system (DBMS) processing a long-running memory-intensive analytical query Q_{lo}. Suppose another memory-intensive query Q_{hi} with much higher priority is now submitted to the DBMS. The result of Q_{hi} may be needed to make a real-time decision, so we need to process Q_{hi} as quickly as possible and with all available resources. Ideally, the DBMS should *suspend* the execution of Q_{lo}, quickly release all resources held by Q_{lo}, and start Q_{hi} using all resources. Query Q_{lo} can be *resumed* once Q_{hi} finishes execution, ideally without losing any significant fraction of the work that Q_{lo} has done prior to suspend. In this paper, we address the problem of implementing query suspend and resume efficiently in a DBMS.

Query suspend and resume are important in many settings:

- *Queries with different priorities*: Today, DBMSs are often used for mixed workloads, with both high-priority, real-time decision-making queries and low-priority, non-customer-facing analytical queries (e.g., OLAP, mining). Typically, DBMSs run low-priority queries whenever high-priority queries are not running. However, with 24×7 operations, the window of time a DBMS can devote fully to low-priority resource-intensive queries becomes smaller and increasingly fragmented. To maximize resource utilization in this setting, it is important for the DBMS to be able to quickly suspend long-running, low-priority queries when high-priority queries arrive; the suspended queries can be resumed when the high-priority ones have completed.
- *Utility and Grid settings*: Queries are now run frequently on computational utilities (e.g., *Condor* [4]) and *Grids* [5] composed of autonomous resources. When the owner of resources wants to use them, queries running on these resources must release control quickly, and migrate to other resources.
- *Software rejuvenation*: Benefits of *software rejuvenation* [7], the practice of rebooting enterprise computing systems regularly, are now recognized widely. Reboot is critical when performance degrades due to resource exhaustion caused by resource leaks. Query suspend and resume is important in this setting because (1) the challenge of predicting query completion times accurately (e.g., see [16]) makes it difficult to schedule query completion to match a rejuvenation schedule; (2) when performance degrades, it may not be cost-effective to wait for all running queries to complete before rebooting the system.
- *DBMS maintenance*: Various maintenance utilities—e.g., statistics update, index rebuild, data repartitioning—need to be run regularly on a DBMS to ensure good performance. Support for suspending and resuming long-running queries gives administrators and self-tuning DBMSs more flexibility in scheduling maintenance utilities.

Query suspend and resume can be handled in different ways. The query Q_{lo} to be suspended can be killed, and restarted during resume. Killing and restarting Q_{lo} wastes time and resources, particularly if Q_{lo} has already performed a lot of work. Furthermore,

*This work is supported by an NSF CAREER Award (IIS-0238386) and by IBM Faculty Awards.

†This work commenced when the author was at Duke University.

SIGMOD'07, June 11–14, 2007, Beijing, China.

a kill-and-restart approach can lead to starvation of the query in a setting where suspend requests are common (e.g., in a shared Grid running on autonomous resources).

Query suspend can be partially achieved in some DBMSs by reducing the priority of the process running Q_{lo}, e.g., using the *renice* command in UNIX. Usually, such approaches affect only the CPU utilization directly, and are ineffective or take a long time to release resources held by memory, I/O, or network-intensive plans. Some DBMSs and operating systems (OSs) provide utilities (e.g., IBM DB2's Query Patroller [10]) to limit the resources allocated to low-priority queries so that high-priority queries will always find free resources. None of these techniques provides a suspend/resume functionality that meets the needs of plan migration, software rejuvenation, and DBMS maintenance.

Another technique to suspend Q_{lo} is to write out its entire in-memory execution state to disk, like an OS *process swap* under heavy memory contention. This technique can have high overhead because queries can easily use gigabytes of memory in modern systems with large memory. For instance, writing 1 GB of data to disk in 1-KB chunks can take up to 90 seconds in a modern system. Writing through storage managers (e.g., *SHORE* [2]) often involves additional system overhead. To make matters worse, in a Grid setting where Q_{lo} needs to be migrated elsewhere, dumping state over the network can have an order of magnitude higher overhead.

Challenges and Contributions We have discussed why we need a better solution for query suspend and resume. In search for this solution, we first point out two well-known ideas on how to deal with the execution state of a data flow for suspend and resume.

- The entire state can be *dumped* to disk on suspend, and read back on resume, like an OS-style process swap. While dumping is simple to implement, it can cause high overhead during both suspend and resume.
- The state can be *checkpointed* at selected points during execution. Suspension keeps track of the exact suspend point; resumption starts from the last checkpoint, rolls forward to the suspend point, and then continues execution. Checkpointing minimizes suspend-time overhead and avoids redoing the work performed up to the last checkpoint. However, checkpointing (1) incurs overhead during execution, and (2) has to redo the work done since the last checkpoint.

Although the general ideas of dumping and checkpointing have been studied in various systems (e.g., [14]), unique challenges and opportunities arise when suspending and resuming DBMS queries, which are complex data flows consisting of physical query operators with well-understood behavior. We next illustrate some of these challenges and opportunities.

Example 1 (Running example). *Figure 1 shows a simple execution plan for $R \bowtie S \bowtie T$ consisting of two* block-based nested loop joins *(*NLJ *for short) [6] and three table scans. We will use this operator tree as a running example in this paper. During each iteration of an NLJ outer loop, the operator first reads output tuples from the outer child to fill a large in-memory* outer buffer*, and then performs joins while reading tuples from the inner child. The content of the outer buffer constitutes the in-memory* heap state *of the NLJ. Figure 2 shows how the amount of heap state changes with time for the two NLJs. The child NLJ first fills up its outer buffer, then its total state plateaus while it reads its inner child and produces $R \bowtie S$ tuples to fill the parent NLJ's outer buffer. When the parent NLJ's outer buffer is full, this operator reads from its inner child and produces $R \bowtie S \bowtie T$ tuples. When the inner child has exhausted its output, the NLJ discards its state and begins rebuilding its outer buffer with the next batch of child tuples.*

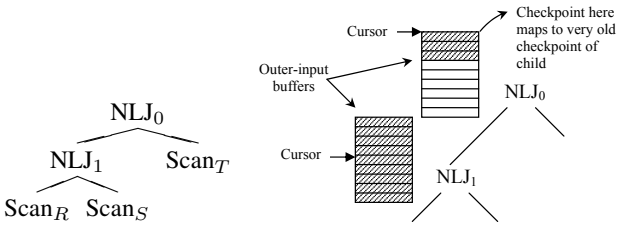

Figure 1: Execution plan and outer buffers.

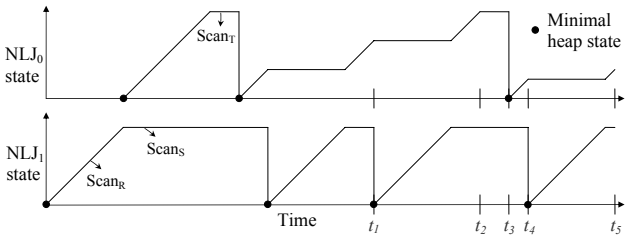

Figure 2: Heap state vs. time, for the two NLJ operators.

Example 2 (Need for asynchronous checkpointing). *Suppose we need to suspend the plan in Figure 1, at the time t_2 depicted in Figure 2. Note that the state of the plan is maximum at this stage, so dumping this state can incur significant overhead during suspend and resume. If we use checkpointing instead, then we would prefer to checkpoint an NLJ when its state is minimum, namely, when its outer buffer is empty. These are indicated as* minimal-heap-state points *for each NLJ in Figure 2. However, notice that the minimal-heap-state points of two operators usually do not coincide.*

In contrast to synchronous checkpointing, which checkpoints the entire query at a single point in time, asynchronous checkpointing poses an interesting challenge: Can we let operators make their own checkpointing decisions, but coordinate independent checkpoints to guarantee that the entire data flow can be resumed to a single consistent state?

Next, consider what happens when a suspend request is received. An operator can choose between dumping its state or *going back* to a previous checkpoint, with the tradeoff discussed earlier. The optimal *suspend plan*, which specifies for each operator whether it should dump or go back, depends on many things—e.g., how soon the suspend must complete, how large is the current state, how much work would be involved in reconstructing it, dependencies among operator states, etc. We argue that different strategies may be appropriate for different operators, and the best suspend plan depends dynamically on the runtime conditions at the suspend time. We illustrate this point with a simple example below.

Example 3 (Need for suspend-time optimization). *Suppose NLJ_1 in Figure 1 produces joining tuples very infrequently, i.e., its join selectivity is low. On a suspend request, therefore, it may be better for NLJ_0 to write its state to disk, rather than to discard these tuples and have NLJ_1 recompute them during resume. Meanwhile, for the same suspend request, it may be better for NLJ_1 to go back to its most recent checkpoint even if its outer buffer is full: Reconstructing NLJ_1's state during resume is relatively inexpensive (just scan R); however, dumping this state to disk is expensive, and it would still have to be read back in during resume.*

We make the following contributions:

- **Query suspend/resume.** We propose a novel DBMS query lifecycle (Section 2) for supporting query suspend/resume. This lifecycle augments the traditional query execution with two new phases—suspend and resume—that are triggered on demand.

(We target scenarios where the DBMS can act on suspend requests as opposed to immediate system failures; nevertheless, we believe some of our techniques can be applied to failure handling.)

- **Semantics-driven asynchronous checkpointing.** We design strategies (Sections 3 and 4) for individual plan operators to decide independently when and how to checkpoint in-memory state. We propose a novel *contract* mechanism to coordinate independent checkpoints in a query plan, and to guarantee that the entire plan can resume to a single consistent state. Our checkpointing techniques incur negligible overhead during execution and allow fast suspend.
- **Online selection of suspend/resume strategies.** We address (Section 5) the problem of choosing the best suspend/resume strategy at suspend time to minimize total suspend/resume overhead given a maximum allowed time to suspend. We solve this optimization problem using mixed-integer programming, show how to implement it efficiently and dynamically, and demonstrate how a *hybrid* strategy—where some operators dump state and others go back—can often be better than a purist approach where the entire query either dumps state or goes back.
- **Planning ahead for suspend/resume.** We show (Section 7) how suspend/resume requirements could influence a query optimizer's choice of execution plan, motivating the use of information about expected suspend requests to select "suspend-friendly" plans that are more efficient overall when suspend and resume costs are included.
- **Implementation.** We show how our techniques can be implemented in a DBMS with iterator-based query execution. We have implemented our techniques in the *PREDATOR* [19] open-source DBMS, and we report experimental results (Section 6) illustrating the benefits of our techniques.

2 Preliminaries and Problem Setup

Iterator-Based Query Execution Query execution plans in most DBMSs consist of a tree of physical operators where each operator implements an *iterator interface* supporting three methods [6, 9]: Open() opens the iterator interface, Close() closes the interface, and GetNext() retrieves the next output tuple from the operator. Query execution proceeds in a recursive pull-based fashion where each operator gets its input tuples by calling the GetNext() methods of its children in the operator tree.

Operators in an iterator-based plan use different types of state during execution. We broadly categorize them as follows.

- **In-memory state**: state retained by an operator in main memory during execution. This state may consists of:
  - **Heap state**: Many operators use in-memory data structures that can be as large as possible. For example, NLJ maintains a large outer buffer for tuples from its outer child. We call such state *heap state* because memory for it is usually allocated separately from the heap. Operators with heap state are called *stateful operators*, e.g., NLJ, sort-merge join, and hybrid hash join [6].
  - **Control state**: *Control state* refers to a small amount of state maintained by an operator to keep track of its execution. For example, NLJ's control state consists of a tuple from its inner child and a cursor over the outer buffer (indicating the current join position). As another example, the control state of a two-phase merge-sort operator consists of locations of the sorted sublists on disk, and cursors over these sublists (if the operator is in the merging phase).

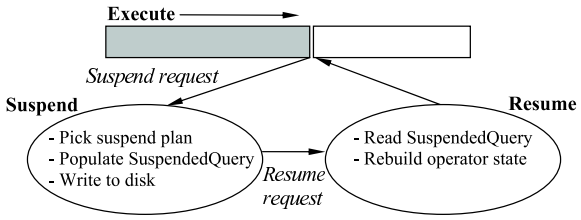

Figure 3: Execute/suspend/resume phases of a query.

- **Disk-resident state**: state retained by an operator on disk, e.g., the sorted sublists produced by a two-phase merge-sort operator.

New Query Lifecycle To support suspend/resume, we augment the standard *execute* phase of a query in a DBMS with two new phases—*suspend* and *resume*—as illustrated in Figure 3. The problem we address is how to support this novel query lifecycle efficiently. Once the query optimizer chooses an execution plan for a query Q, Q enters its execute phase where the plan is instantiated and the iterator-based operators begin executing.

As motivated in Section 1, Q may need to be suspended during execution for a variety of reasons. Upon receiving a suspend request, Q enters the *suspend phase*; see Figure 3. The goal of this phase is to produce a SuspendedQuery data structure that encapsulates all the information needed to resume Q later (i.e., to be able to regenerate Q's execution state at the suspend point). This structure can be written to disk (e.g., if Q will be resumed in the same DBMS) or transferred elsewhere (e.g., if Q will be migrated to a replica DBMS). A *suspend cost* is incurred during Q's suspend phase. After suspend, all of Q's memory resources can be released.

As we saw in Example 3, there are different strategies to suspend each operator in a query plan. At suspend time, the DBMS chooses a *suspend plan* for Q, which describes the strategy that will be used to suspend each operator. The SuspendedQuery structure records both execution and suspend plans for Q. In addition, the suspend plan determines how the rest of SuspendedQuery structure will be populated. For example, a simple suspend plan may specify that all NLJs in the query plan write their entire in-memory state to disk, and record this disk location in SuspendedQuery. Selecting an appropriate suspend plan is a nontrivial problem.

Q enters its *resume phase* when the DBMS is ready to resume it. As illustrated in Figure 3, the goal of this phase is to reconstruct Q's execution state back to the suspend point, so that the execute phase can continue where it was interrupted. The resume phase starts by reading Q's SuspendedQuery structure. Actions during resume are dictated by the suspend plan used during suspend. For example, if the in-memory state of NLJs was written to disk, then during resume we need to read the state back into memory. A *resume cost* is incurred when Q is resumed.

Questions to Answer In this paper we address how to support the novel execute-suspend-resume query lifecycle in a DBMS. Specifically, we tackle the following questions: (1) What is the space of suspend plans for a query? (2) What do we need during execute and resume phases to support such plans? (3) Given a maximum allowed suspend cost, how do we pick the suspend plan that minimizes the total suspend and resume cost? (4) Does it make sense to plan ahead for suspend/resume in query optimization?

Assumptions We make some assumptions: (1) The resumed query uses the same plan and operator-memory allocations as the suspended one (i.e., we ignore the possibility of re-optimizing the query on resume). (2) Both the suspended and resumed query plans see the same physical database state, including the order of tuples already processed. (3) A query plan is executed as a single thread of computation in an iterator-based fashion (e.g., no *exchange* op-

erators [8]). In Section 3.4, we discuss the implications of lifting some of these assumptions.

A Strawman Solution A strawman suspend plan, which we call *all-DumpState*, is one where each and every operator in a query to be suspended follows a *DumpState* strategy. With the DumpState strategy, each operator writes out all its heap state to disk, then adds its control state and the location of the dumped state to SuspendedQuery. If we use all-DumpState in our running example, then each NLJ will dump its outer buffer to disk, and add to SuspendedQuery the location of the dump, the current position of the outer buffer cursor, and the current inner tuple. The table scan operators would record the current positions of their cursors in SuspendedQuery. Finally, SuspendedQuery will be written out to disk.

During resume, using the information from the SuspendedQuery structure written during suspend, the DBMS simply restores the entire in-memory execution state of the query from the disk.

Since all-DumpState is designed along the lines of an OS-style process swap, it is relatively easy to implement. All-DumpState does all its work during the suspend and resume phases, so it adds no overhead during the execute phase.

3 Our Approach

Overview The all-DumpState solution requires every operator to write out its entire in-memory state at the time of suspend, which can take a lot of time to complete and thus delay the release of resources. When there is a need to suspend or migrate a query, the DBMS may have stringent constraints on how quickly it must be done. Therefore, in addition to being inefficient, all-DumpState may be infeasible in certain situations.

As motivated in Section 1, we can periodically checkpoint the execution of a query Q. With periodic checkpointing, suspending Q simply involves writing to disk Q's control state at the suspend point; thus, it can be very fast. When resuming Q, we can start from the last successful checkpoint, roll forward to the suspend point, then continue regular execution.

However, periodically taking *synchronous* checkpoints of the entire execution state of Q adds significant overhead to execution. As we saw in Example 2, operators may be in different stages of execution with very different amounts of heap state. Creating a synchronous checkpoint of Q requires writing a snapshot of Q's entire in-memory state, which can be expensive if Q uses stateful operators like NLJ, sort-merge join, and hybrid hash join, which may hold a large amount of in-memory state in modern systems.

In this section, we describe our comprehensive approach to supporting efficient suspend/resume, consisting of a suite of ideas for all three phases of the query lifecycle:

- In the execute phase, we propose *asynchronous* checkpointing, where each operator checkpoints independently of others in the query plan. We checkpoint stateful operators *proactively*, but only at their *minimal-heap-state points* where the heap state is as small as it gets during execution.
  With these techniques, checkpointing becomes a low-overhead operation; in fact, the amount of state to be remembered is so small that is can be retained in memory and written out quickly at suspend time. Hence, this approach incurs no extra I/Os at all during the execute phase.
- In the suspend phase, thanks to checkpointing, we now have a new suspend strategy as an alternative to DumpState. With this new alternative, called *GoBack*, an operator writes only its current control state to SuspendedQuery at the time of suspend, incurring much lower suspend cost than DumpState.
- In the resume phase, we now need to handle operators differently depending on the strategies chosen by the suspend plan. Heap state handled by DumpState simply needs to be read back; however, heap state handled by GoBack needs to be reconstructed by rolling forward from the last checkpoint, which might result in a higher resume cost than DumpState.
- We consider the tradeoff between DumpState and GoBack in choosing a suspend plan during suspend. We use mixed-integer programming to find the best suspend plan that minimizes the total overhead of suspend/resume while meeting a given suspend-cost constraint.

The remainder of this section presents the details of our approach. We defer the discussion of suspend plan optimization to Section 5. Before delving into the details, we wish to make a few observations that we think are interesting and illustrative of the intricacies involved in implementing suspend/resume:

- Checkpointing each operator separately is insufficient; without cooperation from a child operator to produce the next input tuples, a checkpoint is useless (cf. *contracts* in Section 3.1).
- Simply remembering the latest checkpoint taken is insufficient; a parent operator may want to go back to its minimal-heap-state point, earlier than the latest checkpoint of the child (cf. *active checkpoints* in Section 3.4).
- The above observation does not imply that all checkpoints need to be remembered; in fact, each operator only needs to remember $O(n)$ checkpoints, where n is the number of operators in the query plan (cf. Theorem 1).
- Rolling forward from a checkpoint does not mean we redo all the work in between; some computation can be in fact skipped (cf. Section 3.3).

3.1 Execute Phase

Checkpoints Apart from regular processing in the execute phase, operators create *checkpoints* in anticipation of suspend requests.

Definition 1 (Checkpoint). *A checkpoint for an operator O at time t contains all the information necessary*[1] *to later restore O's execution state as of time t.*

The checkpointing operation for an operator O consists of creating a checkpoint $Ckpt$ and either storing $Ckpt$ in memory or writing it to disk. If $Ckpt$ is stored in memory, then it may need to be written to disk in the suspend phase. Once $Ckpt$ is written to disk, O can use $Ckpt$ in the resume phase to reconstruct its exact state as of the time $Ckpt$ was created.

We do not always know in advance when or whether a suspend request will arrive. Therefore, it is desirable to keep the overhead low for checkpointing during normal execution, particularly for stateful operators like NLJ, sort-merge join, hybrid hash join, etc. Our solution is to have each stateful operator O in a plan create checkpoints proactively at points during execution when O's heap state is minimal, namely, at O's *minimal-heap-state points*. For instance, an NLJ operator will perform a checkpoint each time

[1] In the DBMS query suspend/resume setting, for all the operators we considered, it suffices for the checkpoint to record O's in-memory (heap and control) state. There is no need to copy the disk-resident state at t, because a chunk of such state is usually written once and never modified (e.g., the sorted sublists of a merge-sort operator). The control state records the current locations and sizes for chunks of the disk-resident state; this information goes into the checkpoint, and allows us to restore access to the disk-resident state at the time of the checkpoint. We make this assumption in the rest of Section 3.1.

its outer buffer is empty; NLJ has zero heap state at this point. Minimal-heap-state points are very attractive because:

- Very little bookkeeping is required to create checkpoints at these points: By definition, the heap state is minimal (often zero), while the control state is always small (often implicit at minimal-heap-state points and hence may not need to be explicitly remembered).
- Since the checkpoints are small, and most old checkpoints can be deleted over time (as we will show in Section 3.4), all checkpoints required for a plan can be retained in memory until suspend. Hence, checkpointing incurs no I/Os during execution.

Stateful operators checkpoint their state at every minimal-heap-state point. We refer to this form of checkpointing as *proactive checkpointing*. On the other hand, stateless operators like filters are always at a minimal-heap-state point. Hence, stateless operators perform *reactive checkpointing*, where checkpointing is done only on demand, e.g., on a suspend request. Overall, checkpointing is *asynchronous* across operators because each schedules its checkpoints independently according to its semantics.

Contracts It turns out that checkpoints alone are not enough to support suspend/resume because of asynchronous checkpointing. A checkpoint $Ckpt$ for an operator O can only restore O's execution state *alone*. However, to be able to make progress after resume, O still needs its children in the plan to resume producing input tuples for O immediately after the point where $Ckpt$ was created. To address this problem, after creating a checkpoint, O will establish a *contract* with each of its children.

Definition 2 (Contract). *A contract is an agreement between a parent operator P and a child operator Q. Let $r_1, \ldots, r_n$ be the first-to-last sequence of tuples output by Q to P if they run to completion without being suspended. When a contract Ctr is established between P and Q just before tuple r_i is output by Q, Q agrees to be able to regenerate, at any later point in time when Ctr is enforced by P, tuples $r_i, \ldots, r_n$ in order.*

Next, we discuss what a child operator Q needs to do in order to fulfill a contract Ctr with its parent operator P. We discuss two cases; operational details will be provided momentarily:

- If Q is a stateful operator, then Q saves its current control state when it signs Ctr. (Recall that control state is small and includes, for example, the cursor position for a NLJ.) This control state, along with Q's latest checkpoint (and the contracts with Q's children that Q established at that time), are sufficient for Q to fulfill Ctr.
- If Q is a stateless operator, in order to sign Ctr, Q creates a reactive checkpoint and in turn establishes a contract with its children. These two items are sufficient for Q to fulfill Ctr.

To recap, *whenever the parent operator creates a checkpoint at time t, it has to establish contracts with its children at t.* Thus, an operator's contract is always associated with a parent operator's checkpoint. Furthermore, *to fulfill a contract signed at t, a child operator relies on the last checkpoint of its own created before or at t*—either a proactive checkpoint for a stateful child or a reactive checkpoint for a stateless child. Conceptually, the child first restores its execution state to this checkpoint, rolls forward to t, and starts producing output tuples for its parent in fulfillment of the contract.

Contract Graph To keep track of the dependencies among checkpoints and contracts, we maintain for each query a runtime in-memory data structure called *contract graph*. A contract graph G is a directed acyclic graph with checkpoints as nodes and contracts as edges. Suppose that a parent P creates a checkpoint $Ckpt_1$ and at the same time establishes a contract Ctr with its child Q, while Q signs Ctr and determines that it would rely on its own checkpoint $Ckpt_2$ to fulfill Ctr. At this point, we would record in G a directed edge Ctr from $Ckpt_1$ to $Ckpt_2$.

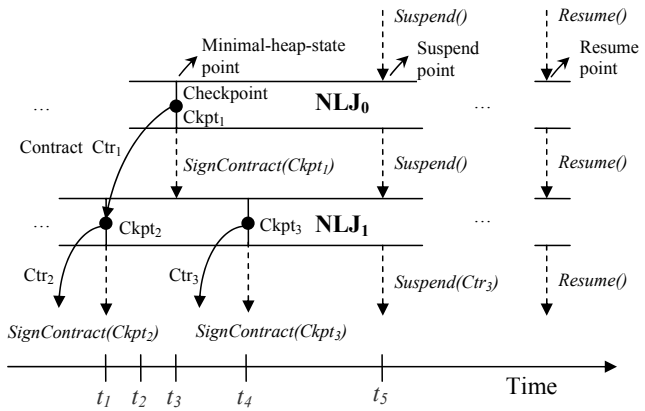

Figure 4: Sequence of calls for two operators.

Checkpoints and contracts become unnecessary over time and may be deleted from the contract graph. We discuss in Section 3.4 how to maintain the contract graph and show that its size is bounded by $O(nh)$ for a plan of n operators with height h.

Operational Details To implement checkpointing and contracting, each operator supplies two methods. Checkpoint() creates a checkpoint; this method is internal to the operator. SignContract($Ckpt$) establishes a contract for the parent's checkpoint $Ckpt$; this method is one of the extensions we propose to the iterator interface.

A stateful operator P calls Checkpoint() at every minimal-heap-state point, which creates a checkpoint $Ckpt_1$ and adds it to the contract graph. Checkpoint() then calls SignContract($Ckpt_1$) on each of P's children. When a child operator Q receives this call, Q creates a new contract Ctr, which stores any control state necessary to be able to resume execution from this point. Then, Q determines its own checkpoint $Ckpt_2$ to use for fulfilling Ctr: If Q is stateful, $Ckpt_2$ is Q's checkpoint at its last minimal-heap-state point; if Q is stateless, $Ckpt_2$ will be created reactively by calling Checkpoint(). Finally, Ctr is added as an edge to the contract graph.

In general, SignContract() calls are recursively propagated down through all stateless children, until some stateful children stop the recursion with their proactive checkpoints.

Example 4 (Checkpointing and contracting in action). *Figure 4 shows the two NLJs from our running example. The marked times $t_1, \ldots, t_5$ correspond to the times shown in Figure 2. NLJ_0 creates a checkpoint node $Ckpt_1$ at its minimal-heap-state point t_3 and invokes SignContract($Ckpt_1$) on NLJ_1. NLJ_1 responds by creating a contract Ctr_1 which stores the current control state of NLJ_1. Ctr_1 is mapped as the edge between nodes $Ckpt_1$ and $Ckpt_2$, where $Ckpt_2$ is NLJ_1's proactive checkpoint at its last minimal-heap-state point t_1. $Ckpt_1$, $Ckpt_2$, and Ctr_1 form the corresponding contract graph. We also show NLJ_1's SignContract() calls. Figure 5 shows a more complex contract graph for four stateful operators evolving over time, and is explained in Section 3.4.*

3.2 Suspend Phase

Choosing a Suspend Plan When the DBMS wants to suspend a query Q, it raises a *suspend exception* in the thread that runs Q. The exception is handled at Q's next blocking step (e.g., I/O) and Q enters the suspend phase where its first task is to choose a *suspend plan*. A suspend plan specifies one of two possible actions—DumpState or GoBack—for each operator. Identifying the space

of valid plans and choosing the best one are nontrivial problems that we address in Section 5. For now, we will assume that a valid suspend plan has been chosen.

Carrying Out a Suspend Plan Given the suspend plan, the suspend exception handler initializes the SuspendedQuery data structure with the execution and suspend plans for the query Q to be suspended. Then, the handler invokes a method called Suspend() on the root query operator, which will further populate SuspendedQuery with additional information necessary for later resume.

On receiving the Suspend() request, the root query operator O_r looks up the suspend plan for the action specified for it:

- If the action is DumpState, then O_r writes its heap state to disk, and records its disk location in SuspendedQuery. In addition, O_r adds its current control state—needed to resume from exactly the suspend point—to SuspendedQuery. Finally, O_r calls Suspend() on each of its children so that the next GetNext() call (on resume) will retrieve the tuple following the last one received by O_r.
- If the action is GoBack,[2] O_r finds $Ckpt$, its latest checkpoint; then, O_r adds both the content of $Ckpt$ and its current control state to SuspendedQuery. Instead of calling Suspend() on a child O_c, O_r calls Suspend(Ctr), which basically instructs O_c to suspend to an earlier point. Here, Ctr is the contract that O_r established with O_c when creating $Ckpt$. Effectively, in this case, O_r discards its heap state and depends on its children to reconstruct its state at resumption.

The Suspend() call on the root query operator eventually causes a Suspend() or Suspend(Ctr) call on each non-root operator P in the query plan. P processes a Suspend() call in exactly the same fashion as the root operator. When Suspend(Ctr) is called on P, it looks up the suspend plan to determine its action:

- In case of DumpState, P writes its heap state to disk, and records its disk location in SuspendedQuery. In addition, P adds the content of Ctr (the control state recorded earlier when signing Ctr) to SuspendedQuery. This information will allow P to resume execution from the point when contract Ctr was signed. Finally, P calls Suspend() on its children.
- In case of GoBack, P first adds the content of Ctr to SuspendedQuery. Let $Ckpt$ be the checkpoint of P pointed to by the edge for contract Ctr in the contract graph. For each child Q, P calls Suspend(Ctr_Q) on Q where Ctr_Q is the contract between P and Q established at P's $Ckpt$. Effectively, P instructs all of it children to resume (later in the resume phase) to $Ckpt$; from that point, the subplan rooted at P can together roll forward to the time of Ctr.

Finally, after all Suspend() or Suspend(Ctr) calls have been made on all operators in the query plan, the suspend exception handler writes out SuspendedQuery to disk and stops Q, discarding all its in-memory state and completing its suspend phase.

Example 5 (Suspend in action). *Suppose suspension occurs at time t_5 in Figure 2. At this point, the NLJ outer buffers are as shown in Figure 1. The corresponding suspend point is also indicated in Figure 4. Assume that the suspend plan selected is one where NLJ_0 does DumpState and NLJ_1 does GoBack. First, Suspend() is called on the root operator NLJ_0. Since NLJ_0 chooses DumpState, it writes the few tuples in its outer buffer to disk, then records the disk location of this state as well as its current control state in SuspendedQuery. It then calls Suspend() on its child operators as seen in Figure 4. NLJ_1 chooses GoBack, so it just adds its current control state to SuspendedQuery (NLJ checkpoints at minimal-heap-state points happen to contain no information). Finally, NLJ_1 invokes the previous contract established at time t_4 with its child operator by calling Suspend(Ctr_3) on $Scan_R$. $Scan_R$ adds the content of Ctr_3 (which contains its control state at t_4) to SuspendedQuery so that it can regenerate all its output tuples starting from t_4.*

[2]We only consider a stateful operator in this case, because DumpState would work better for a stateless operator.

3.3 Resume Phase

We now discuss steps required when resuming a suspended query. First, the SuspendedQuery data structure is read back from disk. The query plan is recreated by instantiating all operators, and the Resume() method is invoked on the root operator. Eventually, Resume() gets called on all operators, causing the plan to get ready to produce the tuple immediately after the last one produced before suspend. When the root operator receives the call, it first calls Resume() on its children to get them into the correct state. It then looks up the SuspendedQuery structure and either loads its internal state from disk (in case of DumpState) or calls GetNext() appropriately on its children (in case of GoBack) to recompute its internal state. In the latter case, each child will produce the correct tuples because we previously invoked its Resume() method.

Example 6 (Resume in action). *In our running example (cf. Figure 4), Resume() is called recursively on all operators. Assume that at suspend time, the online optimizer chose the suspend plan described in Example 5. (1) $Scan_R$ reads in the control state and gets into a position where the next tuple generated would be the one immediately after checkpoint $Ckpt_3$. (2) NLJ_1 reads from SuspendedQuery the control state for t_5, which is the "target state" that NLJ_1 should roll forward to. Then, NLJ_1 starts with an empty outer buffer and calls GetNext() on $Scan_R$ to refill the buffer. (3) NLJ_0 directly reads its heap and control state from SuspendedQuery. These actions put the query plan in a position capable of producing tuples starting precisely from the suspend point, as desired.*

Skipping versus Redoing During the resume phase, it is important to point out that an operator using GoBack strategy to a checkpoint $Ckpt$ does not recompute all its output tuples from $Ckpt$ to the suspend point. In fact, it only needs to redo the work of generating its output tuples from Ctr on, where Ctr is the contract used (if any) in the Suspend(Ctr) call to the operator. Output tuples from $Ckpt$ to Suspend(Ctr) are *skipped* instead. If there is no such Ctr (i.e., the operator received Suspend() directly), then all output tuples to the suspend point can be skipped.

Skipping can be done efficiently knowing the operator's "target state" at either the suspend or contract point, which is available from the SuspendedQuery structure. We illustrate this point using an NLJ operator that was suspended in the middle of joining an inner tuple with tuples in its outer buffer. Suppose that the suspend plan chooses GoBack for this NLJ. On resume, the NLJ refills its outer buffer from its outer child. Then, the NLJ directly restores the target state: the inner tuple and the position of the outer buffer cursor. The next output tuple to be generated will be precisely the one after the suspend point. There is no need to recompute (and discard), for example, the join between the inner tuple and the outer tuples before the target cursor position.

Suspend During or After Resume If a new suspend request comes *during* resume, we can simply discard everything and keep the old SuspendedQuery data structure; the next resume will restart using the same data. If a suspend request comes *after* the resume phase completes, we have two choices. If we save the contract graph to disk on each suspend, then we will have full flexibility in performing another suspend soon after resume. This option is feasible since

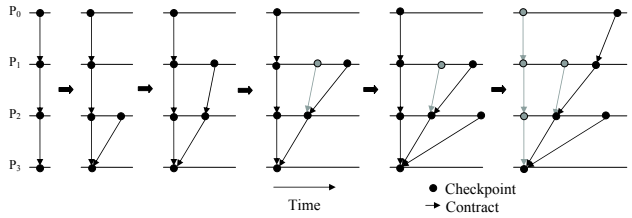

Figure 5: Contract graphs over time.

contract graphs are typically small (see Section 3.4). If we do not store the contract graph, part of the contract graph (a subtree) is still available in the SuspendedQuery data structure. With a partial graph, we will not have as many options for suspend plans as we could; however, as the query execution continues, the contract graph will be gradually reformed to enable more options.

3.4 Discussion

Contract Graph Maintenance Operators create checkpoints and contracts during their execution. As time progresses, older checkpoints and contracts become useless because we no longer need them during suspension, so they can be deleted. Formally, an *inactive checkpoint* is one to which no operator will go back during suspension, and an *inactive contract* is one which will never be enforced during suspension. Inactive contracts and checkpoints can be deleted from the contract graph. When an operator creates a new checkpoint, it checks if any of its previous checkpoints can be deleted. A checkpoint node in the contract graph can be deleted if it has no incoming edges and it is not the most recent checkpoint of some operator. If a checkpoint node can be deleted, it is removed along with its outgoing edges (contracts). The child nodes of deleted checkpoints are recursively checked to see if they can in turn be deleted.

Example 7 (Contract graph maintenance). *Consider a left-deep query plan consisting of four NLJ operators named P_0, P_1, P_2, and P_3 starting from the root. The leaf operators are table scans. A possible evolution of contract graphs with time is shown in Figure 5 (we only show the NLJ operators for clarity). Initially, all four operators create checkpoints at the same time (just before execution starts), giving the first contract graph. P_2 is the first to reach its next minimal-heap-state point, and it results in a new checkpoint node and contract edge as shown in the second frame. Next, P_1 creates a new checkpoint node. The next minimal-heap-state point again belongs to P_1, and at this time the earlier checkpoint node and contract edge can be deleted (shown shaded in the figure). Next comes P_2's checkpoint and finally, when P_0 creates a new checkpoint, an earlier set of checkpoints and contracts can be deleted.*

The theorem below follows directly from the definitions of inactive checkpoints and contracts:

Theorem 1. *For a query plan consisting of n operators where h is the tree height of the plan, the maximum number of checkpoints and contracts in an actively maintained contract graph is $O(nh)$ (or $O(n^2)$ in the worst case).*

Proof. (Sketch) Consider an operator P. An active checkpoint of P must be either the last one for P, or reachable via contract edges from an ancestor checkpoint that is the last one for its corresponding operator. There are at most h such ancestor checkpoints, each of which has only one descendant checkpoint for P. Thus, the number of active checkpoints for each operator is no more than h. The number of active contracts are on the same order as the number of active checkpoints. □

Recall that we only create checkpoints at minimal-heap-state points, or reactively for stateless operators, the space required for each checkpoint is small. Contracts only record control state, which is also very small. Since the number of operators in a query plan is usually not huge, the overall space taken by the entire contract graph is typically no more than a few kilobytes in size.

Contract Migration This technique reduces resume cost by migrating contracts signed by an operator to later points during execution. We describe two scenarios where migration is useful. (1) Consider a stateful operator P such as an NLJ. If a contract has been established between P's parent and P at some point, and P reaches the next minimal-heap-state point without generating any tuples (maybe because the join produced no matches), the older contract can be safely migrated to the new checkpoint of P. (2) Similarly, a very selective filter operator can avoid re-performing the work of reading non-matching tuples from its child after signing a contract with its parent. This optimization can be achieved by creating a new reactive checkpoint after obtaining the first matching tuple, and migrating the old contract to this checkpoint.[3]

Generalizing Suspend Plan Consider an operator such as merge join, whose heap state is derived from two or more children. It may be optimal for this operator to choose different strategies with respect to each child. Based on this intuition, we can extend the definition of a suspend plan as follows. A suspend plan specifies, for each operator, the strategy to be used with respect to each of its children. For example, a merge join operator may decide to choose GoBack w.r.t. its left child and DumpState w.r.t. its right child. Extending our earlier discussion to support general suspend plans is straightforward; we omit the details due to space reasons.

Resuming with a Different Plan Resuming with a different plan is tricky because we exploit physical properties of operators for efficiency. For instance, changing buffer size could change intermediate result ordering, and could mess up the resumption of ancestors. We can (with modifications) handle changes to operator-memory allocations on query resumption; details are omitted.

Resuming with a very different plan is attractive as that would make our approach useful for query re-optimization. However, this is not our intended application (Section 8 has more details).

To provide resume flexibility, one extreme would be a purely logical approach where we semantically describe the partial work completed at suspend time, e.g., "all sales in North America have been aggregated." The remaining work can be cast as a distinct, independently-optimized query as in [13]. However, it is difficult to capture the partial work done at arbitrary points in time, so the opportunity for applying the logical approach is severely limited. A logical approach cannot exploit the knowledge of the internals of query plans and operators like our approach.

Thus, there is an inherent trade-off between the flexibility in resume and applicability/efficiency. Instead of being limited to a single solution, the DBMS can incorporate multiple approaches that are used for different scenarios.

Assumption on Same Database State Our assumption that the suspended and resumed query plans see the same database state (Section 2) does not imply a read-only database, because we only need to block writes to those portions of the database that were read by an unfinished query. When the query completes and releases its

[3]One technicality is that, since the matching tuple has already been returned by the filter's child, it might be too late to ask the child to reproduce that tuple. The filter would therefore save this single tuple as part of the migrated contract. On resume, the saved tuple would be the first one returned by the filter.

locks, writes to these portions can proceed. This behavior is no different from a standard DBMS running serializable transactions. Suspend/resume does not introduce new restrictions. Techniques that benefit a standard DBMS, such as predicate locking, also benefit suspend/resume. Finally, if a high-priority update conflicts with a suspended query, we can always terminate (and later restart) the suspended query.

4 Implementation for Other Operators

So far, we have used mostly block-based NLJ as an example in discussion. We now briefly discuss some of the other common operators. Additional operators, including index NLJ, simple hash join, hybrid hash join, grouping with aggregation, and duplicate elimination, are discussed in [3] due to space constraints.

Table Scan and Index Scan Table and index scans are the base operators upon which query plans are built. Table scans operate over disk pages, returning the next tuple from the retrieved disk page in response to GetNext(). Index scans are similar, except that they operate over an index structure.

- Contracting: Being a leaf operator, table scan performs only reactive checkpointing. When it receives a SignContract($Ckpt$) request from its parent, it creates a contract which stores the current control state (i.e., the current disk page location and position within that disk page) in order to satisfy that contract.
- Suspend: On Suspend(), table scan writes its control state to the SuspendedQuery data structure. On a Suspend(Ctr) call, table scan retrieves the control state from Ctr and writes it to SuspendedQuery.
- Resume: Resume() determines the location of the disk page to be retrieved, from SuspendedQuery. It reads in the disk page to memory and positions the cursor at the right location.

Filter The filter operator works by consuming tuples from its child and producing a subset of tuples that pass the filter condition.

- Contracting: As a stateless operator, filter performs only reactive checkpointing. When it receives a SignContract($Ckpt$) request from its parent, it creates an empty contract to satisfy the request and then establishes a contract with its own child. Filter can use contract migration (see Section 3.4) to avoid reperforming some extra work at resumption.
- Suspend: Filter is stateless, so it responds to Suspend() by calling Suspend() on its child, and to Suspend(Ctr) by calling Suspend(Ctr') on its child, where Ctr' is the corresponding contract between the filter operator and its child.
- Resume: Filter simply calls Resume() on its child. In case of contract migration, it reads one tuple from SuspendedQuery.

Merge Join Merge join takes two sorted inputs and produces the join output. Merge join calls GetNext() on each child to form batches of tuples with equal join attributes (called value packets [9]). Although value packets simplify join logic, they introduce state that we need to consider. The DumpState strategy requires writing out and reading in both the current value packet (heap state), and the cursor positions within the value packet (control state). Minimal-heap-state points occur when a value packet is exhausted and proactive checkpointing can be performed at these points. On receiving a SignContract($Ckpt$) call, the operator needs to store its control state in the newly created contract. Contracts would ensure regeneration of the current value packet. If value packets are small, we could instead perform reactive checkpointing and store the value packet as part of the contract with parent.

Two-Phase Merge Sort During its first phase, this operator successively reads tuples into its sort buffer, sorts them with an in-memory sorting algorithm, and then writes the sorted sublist back on disk. During the second phase, it buffers one block from each sublist, and produces tuples by consuming the minimum first tuple, refilling blocks as necessary. The sorted sublists that have been written to disk are disk-resident state; they serve as an convenient *materialization point* [17], and can be retained across suspend and resume (assuming plentiful disk space).

During the first phase, the sort operator performs proactive checkpointing before reading each new sublist into the sort buffer. It receives a SignContract($Ckpt$) only once at the beginning of this phase because this phase does not produce tuples. Contract migration is crucial and done at every proactive contract. On a Suspend() or Suspend(Ctr) call, the DumpState strategy requires writing out the unsorted sublist (which may have been partially read) from heap to disk and calling Suspend() on the child. GoBack requires the operator to call Suspend(Ctr') where Ctr' is the contract with child. During the second phase, sort behaves similarly to a table scan.

5 Selecting Suspend Plans Online

In this section we address the problem of choosing the optimal suspend plan at suspend time. Every operator needs to decide whether to choose the DumpState or the GoBack strategy. Choosing GoBack often reduces, by orders of magnitude, the state to be written to the disk at suspend time. The downside is a potentially longer resume, because the discarded state needs to be recomputed by the subplan. Since high-priority tasks may be waiting, we would like make suspend as fast as possible, at the expense of a slower resume. However, within a specified suspend budget, we may still write out state to disk if it results in a lower overall overhead.

Given a constrained suspend cost budget, we address the optimization problem that determines, for a plan of n operators, the suspend strategy for each operator. The time of suspend is the ideal time to perform this optimization, because we have all correct statistics necessary, and we know the exact position of suspend and the position of each operator with respect to its contracts and checkpoints. The optimization problem can be formulated as a mixed-integer program.

Space of Valid Suspend Plans As a slight simplification, we assume that we make suspend choices on a per-operator basis. In general (see Section 3.4), an operator could choose different strategies for each of its children. The extension is straightforward.

It would then appear that there are 2^n possible suspend plans, since each operator can choose either DumpState or GoBack. However, there are restrictions on what combinations are valid. If a parent chooses the GoBack strategy, it could imply that the child also has to use GoBack, if the child has been requested to satisfy a contract that it accepted before its last checkpoint. In this case, the operator would have discarded the relevant state and hence it has to use GoBack and depend on its contract with its child to regenerate that state. Such restrictions can be determined at runtime based on the state of the operators at suspend time.

Mixed-Integer Programming Formulation We have n operators, $p_1 \ldots p_n$ forming an operator tree. We define the set $anc(i)$ as the set of ancestors of p_i and p_i itself. Let $par(i)$ denote the parent of p_i in the tree. We define the following constants:

- $d_i^s \; \forall i$: Operator-specific suspend cost for p_i if p_i chooses DumpState; mostly the cost of writing p_i's current state to disk.
- $d_i^r \; \forall i$: Operator-specific resume cost for p_i if p_i chooses DumpState; mostly the cost of reading the dumped state.

- $c_{i,j}$ $\forall i, j$ s.t. $j \in anc(i)$: A constraint on whether p_i can choose DumpState: $c_{i,j}$ is 1 if p_i's most recent checkpoint is *after* the checkpoint of p_i that is reachable (in the contract graph) from the latest checkpoint of p_j; 0 otherwise.
- $g^s_{i,j}$ $\forall i, j$ s.t. $j \in anc(i)$: Operator-specific suspend cost for p_i if p_i chooses GoBack to fulfill the contract originally initiated by p_j; usually negligible.
- $g^r_{i,j}$ $\forall i, j$ s.t. $j \in anc(i)$: Operator-specific resume cost for p_i if p_i chooses GoBack to fulfill the contract originally initiated by p_j. This cost is approximated by tracking the cumulative work that p_i itself had done up to each of its active checkpoints. $g^r_{i,j}$ is just the difference between the current cumulative work done by p_i and the cumulative work done by p_i at its checkpoint reachable (in the contract graph) from p_j's last checkpoint.
- C: The total suspend budget available.

The variables to be assigned are:

- $x_{i,j}$ $\forall i, j$ s.t. $j \in anc(i)$: For each operator p_i, we define one zero-one variable for each operator p_j in the set $anc(p_i)$. $x_{i,j} = 1$ if p_i decides to go back to fulfill the latest contract originally initiated by p_j; 0 otherwise. If all variables of a particular operator are 0, the operator chooses to dump its state.

The linear program minimizes:

$$\sum_i \Big(d^s_i \big(1 - \sum_{j \in anc(i)} x_{i,j}\big) + \sum_{j \in anc(i)} g^s_{i,j} x_{i,j} \Big) + \quad (1)$$

$$\sum_i \Big(d^r_i \big(1 - \sum_{j \in anc(i)} x_{i,j}\big) + \sum_{j \in anc(i)} g^r_{i,j} x_{i,j} \Big) \quad (2)$$

Subject to:

$$\sum_{j \in anc(i)} x_{i,j} \leq 1 \quad \forall i; \quad (3)$$

$$x_{i,j} \leq x_{i',j} \quad \forall i, i', j \text{ s.t. } i' = par(i) \text{ and } j \in anc(i'); \quad (4)$$

$$x_{i,i} \leq 1 - \sum_{j \in anc(i')} x_{i',j} \quad \forall i, i' \text{ s.t. } i' = par(i); \quad (5)$$

$$x_{i,j} \geq x_{i',j} \text{ if } c_{i,j} \; \forall i, i', j \text{ s.t. } i' = par(i) \text{ and } j \in anc(i'); \quad (6)$$

$$\sum_i \Big(d^s_i \big(1 - \sum_{j \in anc(i)} x_{i,j}\big) + \sum_{j \in anc(i)} g^s_{i,j} x_{i,j} \Big) \leq C; \quad (7)$$

$$x_{i,j} \in \{0, 1\} \quad \forall i, j \text{ s.t. } j \in anc(i). \quad (8)$$

The objectives on Lines (1) and (2) compute the total suspend and resume costs respectively. The constraint on Line (3) ensures that an operator can either choose DumpState (sum would be 0) or it can go back to exactly one ancestor P (sum would be 1). P is either the root or the closest ancestor whose parent chooses DumpState. Lines (4) and (5) basically ensure that the strategies chosen by a parent and a child are compatible. Line (6) ensures that if an operator P's parent chooses to GoBack to some operator Q, then P has to GoBack to Q if DumpState is not viable. Line (7) checks the suspend budget, while Line (8) specifies variable domains.

The total number of constraints is $O(nh)$ (or $O(n^2)$ in the worst case), where h is the height of the plan tree. The solution is used to decide the suspend strategy for each operator. In practice, we find the optimization to be very efficient, taking a time of less than 60ms for plans with up to 100 operators (see [3] for details).

6 Experiments

In order to evaluate query suspend and resume, we implemented our techniques in the PREDATOR [19] database system. We extended the operator interface to include our methods of SignContract(*Ckpt*), Suspend(), Suspend(*Ctr*), and Resume(). We implemented checkpoints and contracts for the most common physical operators including block-based NLJ, sort-merge join, two-phase merge sort, filter, and table scan.

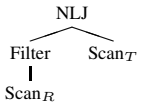

Figure 6: NLJ$_S$ plan.

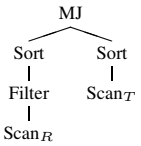

Figure 7: SMJ$_S$ plan.

We implemented the GoBack and DumpState strategies for each operator. We implemented the online optimizer to choose the optimal strategy plan at suspension time, by incorporating a mixed integer program solver into PREDATOR. The query processor maintains all the necessary statistics, and generates the linear program at the time of suspend. The solution is used to generate the optimal suspend plan. We found that even for large query plans, the solver has negligible runtime overhead. PREDATOR uses SHORE [2] as the underlying storage manager. For uniformity, we also used SHORE to write/read data and state. The experiments were run on an Intel Xeon 2.00 GHz machine running Linux kernel 2.6.16. The database was located on a local disk accessed through the SHORE interface. In order to test various operator trees under controlled conditions, we allow the user to specify the physical plan to be used in executing a query.

In addition to our online optimizer strategy (called LP), we experiment with two suspend plans: (1) all-DumpState, where all operators follow DumpState, and (2) all-GoBack, where all operators perform checkpointing and follow GoBack. These suspend plans illustrate the possible performance impact of choosing the right strategy. Although we do not explicitly show the performance of the technique where the system performs periodic synchronous checkpointing and dumping state of all operators (its performance is highly dependent on parameters such as the frequency of the dumping operation), it can easily be seen that in terms of total overhead, all-DumpState corresponds to the ideal hypothetical implementation of such a strategy, where an oracle performs checkpointing and dumping just once, i.e., just before suspension. In practice, due to the regular periodic dumping at synchronous checkpoints, the overhead of this periodic synchronous checkpointing strategy would be much higher than our approaches.

We compare the strategies over two metrics: (1) Total overhead time, which is the total amount of extra work done due to query suspend and resume, and (2) Total suspend time, which is the total amount of extra work done by the system at the time of suspend. The latter metric is important because there may be constraints on the available budget at suspension time.

6.1 Simple Query Plans

In this subsection, we use two simple query plans to experiment with the performance of various strategies. Simple plans allow us to vary various query parameters in a controlled manner in order to analyze their effect on performance.

The first plan NLJ$_S$ (see Figure 6), consists of a block-based NLJ operator with a filter as the left child. The outer buffer size is $200,000$ tuples. The second plan SMJ$_S$ (see Figure 7) has a merge join (MJ) operator. Both its children are two-phase merge sorts; sort buffer size is $200,000$ tuples. The left sort operator has a filter as its child. In both tables R and T, each tuple is 200 bytes long, with an integer key. Table R is populated with 2.2 million unique tuples, with random unique integer key values.

Effect of Filter Selectivity and Suspend Point We first show the effect of filter selectivity (which determines recomputation cost), on the performance of different suspend plans. We first use the NLJ$_S$ plan with varying filter selectivity. The suspend point occurs halfway through filling the outer buffer of NLJ, i.e., after reading

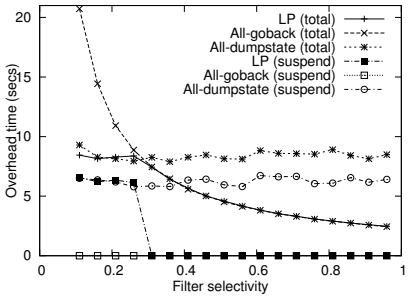

Figure 8: NLJ$_S$, varying selectivity.

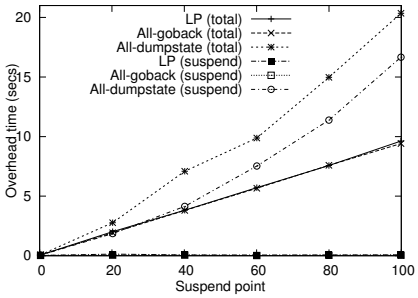

Figure 9: SMJ$_S$, varying suspend point.

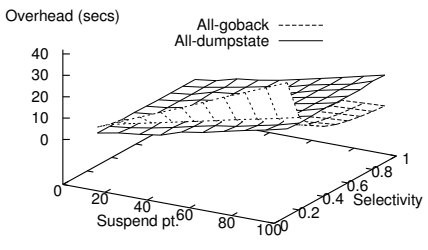

Figure 10: NLJ$_S$, varying selectivity and suspend point.

100, 000 tuples from the child. The results are shown in Figure 8. When the selectivity is very low, it is costly to fill up the outer buffer of NLJ with 100, 000 tuples. On a suspend at this point, the all-GoBack does not do well because the work has to be redone on resume. On the other hand, all-DumpState does better because the work is not lost as it is written out to disk at the time of suspend. Writing in SHORE is more expensive than reading, and hence all-GoBack does better in terms of total query overhead for selectivities greater than around 0.28. In terms of overhead at suspend time, however, all-GoBack is always much better, because it needs to write out only a few bytes of control state as opposed to a large amount of heap state in case of all-DumpState. For this plan, there are no other better hybrid suspend plans, and the optimal suspend plan is one of these two, depending on the filter selectivity. We see that our online LP strategy always selects the best suspend plan, i.e., for low selectivities it selects all-DumpState while for high selectivities it selects all-GoBack (suspend budget is unlimited in this experiment). The same experiment was repeated with the SMJ$_S$ plan, and the results (not shown) follow a similar trend.

We next vary the suspend point for a plan, so that suspension occurs after some percentage of the operator's buffer has been filled. Figure 9 shows the result for the SMJ$_S$ plan (NLJ$_S$ was found to give similar results). The x-axis varies the percentage of the sort buffer filled up at suspension. Selectivity is fixed at 0.5. The suspend point determines the amount of state to be written out in case of DumpState, or the amount of state to be recomputed in case of GoBack. For this simple plan, if one strategy is better than the other for a particular selectivity, it will remain better regardless of the suspend point (at that selectivity). However, the difference between these two strategies will increase as we move the suspend point towards the end of the buffer. At selectivity 0.5, since GoBack is better than DumpState, we see the expected trend. This experiment shows that the strategy choice becomes more vital if suspend occurs when an operator has greater state in memory. Again, online LP always selects the best suspend plan.

Finally, Figure 10 displays the surface plot of total query overhead time for all-GoBack and all-DumpState, over suspend points and selectivities for NLJ$_S$. Suspend point is again measured as the percentage of NLJ's outer buffer filled up at suspend time. The plot matches expected behavior: increasing selectivity changes the preferred strategy, while increasing suspend point location within the buffer exacerbates the difference between the strategies.

Advantage of Suspend-Time Optimization In this experiment, we highlight the importance of making decisions at suspend time by comparing the performance of the online optimizer against one that uses offline statistics to make a strategy choice. We use the NLJ$_S$ plan as before. This time, the R table is filled with around 3 million tuples. However, the data in the table has a skewed distribution such that for the initial (approximately) two-thirds of the table, the filter selects only 1 in 10 tuples. For the remaining portion of the table, the same filter selects 9 out of every 10 tuples. The effective selectivity over the entire table is 0.385. From Figure 8, we see that GoBack is better than DumpState for selectivities greater than around 0.28. Hence, a static optimizer that uses table-level statistics to determine the suspend plan would choose all-GoBack for this query. However, this suspend plan is not good for the portions of the table where the dynamic selectivity is very low.

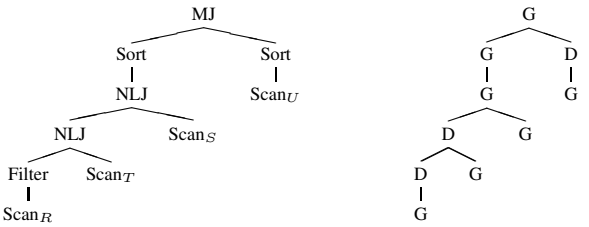

Figure 11: Complex plan and optimal strategy.

We perform suspension at various points of the outer table scan, and plot the total query overhead time and suspend time for both the static and the online LP suspend plans, against the suspend point (expressed in terms of number of R tuples after which suspension occurs). The results are shown in Figure 12. We see that since our online optimizer uses runtime statistics to decide the strategy, it correctly chooses the all-DumpState suspend plan when suspension occurs in the first part of the table. Next, it chooses the all-GoBack suspend plan in the second part of the table as expected. The offline strategy, on the other hand, is unable to adapt dynamically.

6.2 Complex Query Plans

In previous experiments, the optimal suspend plan was one of the extremes i.e. all-GoBack or all-DumpState. In this set of experiments, we examine more complex query plans where the optimal suspend plan may be neither of these two extremes.

Suspend Plan and Performance We create a complex plan involving 10 operators, as shown in Figure 11 (left). Table R contains 2.2 million tuples, while the selectivity of the filter is set to 0.1. The outer buffers of NLJ and the sort buffers have a size of 200, 000 tuples. We suspend query execution when the upper NLJ operator is around 85% full. The online optimizer chooses the suspend plan shown to the right of Figure 11. This optimal plan is neither of the two extremes (all-GoBack and all-DumpState). It is a combination of different strategies for different operators.

In Figure 13, we compare the performance of the online suspend plan against all-GoBack and all-DumpState, in terms of total query overhead and overhead at suspension. We see that the online approach using the hybrid suspend plan is able to perform much better than the purist techniques.

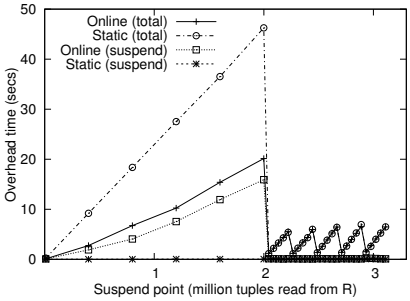

Figure 12: NLJ$_S$, online vs. offline strategies.

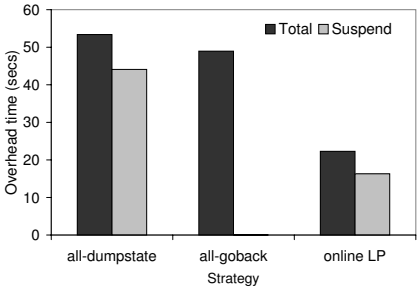

Figure 13: Complex plan, comparison of approaches.

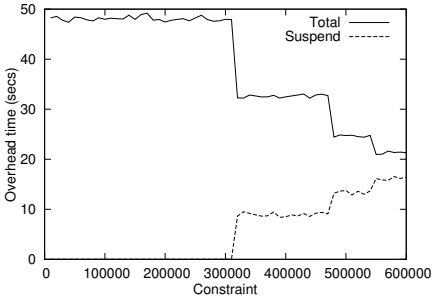

Figure 14: Complex plan, varying suspend constraint.

Varying Constraint on Suspend In this experiment, we increase the available suspend budget and examine how the online optimizer chooses suspend plans. We use a left-deep plan with 3 block-based NLJ operators, and a filter of selectivity 0.1. The NLJ operators have different outer buffer sizes. Figure 14 shows the total overhead and suspend overhead of the online optimizer's chosen suspend plan, as we increase the suspend budget (which we measured as a function of I/O read and write cost, and is approximately proportional to time). For low budgets, the optimizer is forced to choose all-GoBack, leading to high total overhead. As we increase the constraint, the optimizer is able to choose better plans involving a mix of GoBack and DumpState strategies. Thus, the total overhead reduces with a corresponding increase in suspend time (within the provided constraint). Finally, when the suspend budget permits, it switches to the optimal suspend plan with minimal query overhead.

7 Planning Ahead for Suspend/Resume

A standard DBMS query optimizer is entrusted with the task of choosing a near-optimal execution plan for a given query, based on expected execution cost. However, some query plans may be more amenable to suspend/resume than others and can perform better in the presence of suspend requests. We argue that if the plan was chosen without any consideration of suspend/resume at runtime, the plan may be suboptimal. If we know the expected pattern of suspend requests, we should choose a query plan tailored for such a situation. The following examples serve to motivate suspend-aware query optimization. We leave the techniques for choosing the best execution plan with suspend as future work.

Example 8. *Consider a join between tables $R(a, b)$ and $S(c, d)$:*

```
SELECT * FROM R,S WHERE R.a<100 and R.b=S.c;
```

R has $2,200,000$ tuples while S has $250,000$ tuples. The selectivity of the filter predicate is 0.1; i.e., around $220,000$ R tuples are returned by the filter immediately above the table scan for R. Main memory can hold $150,000$ tuples. Let 100 tuples fit on a disk page.

Assume that we have two possible query plans. One uses a hybrid hash join (HHJ) that builds the in-memory hashtable on R. The second uses a sort-merge join (SMJ) with a sort buffer of size $150,000$ tuples. Figure 15 shows the analytical performance of these two plans (in terms of number of disk I/Os) with and without suspends. If there are no suspends, HHJ is better, and is chosen by the optimizer. However, if a suspend occurs, say during the last phase of join, SMJ performs much better.

Example 9. *Consider the same query from Example 8. Now, R has $300,000$ tuples while S has $350,000$ tuples. The selectivity of the filter predicate is 0.6 (i.e., $180,000$ R tuples are returned by the filter). Assume that table S is already sorted on c.*

Assume that we have two possible query plans. One uses an NLJ with an outer buffer that fits $90,000$ tuples. The optimizer would choose R as the outer table, and the total cost in terms of I/Os, ignoring the cost of writing out the result, would be $3000 + 2 \times 3500 = 10,000$ (we would need to scan the S table twice). If we instead use a sort-merge join (SMJ) with a sort buffer that holds $10,000$ tuples, the total cost would be $10,100$ I/Os (read $300,000$ R tuples, write $180,000$ sorted R tuples in sublists, read $180,000$ sorted R tuples during second phase of sort, and read $350,000$ presorted S tuples). Hence, the optimizer would choose NLJ.

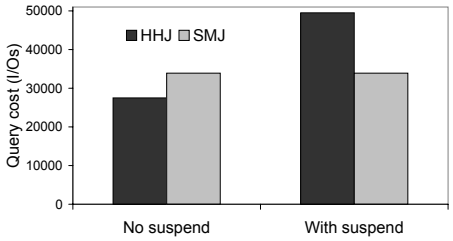

Figure 15: HHJ vs. SMJ; with/without suspend.

Now, assume that we know suspends will occur. If a suspend occurs when the outer buffer in the NLJ plan has filled $80,000$ tuples, the overhead of suspend and resume assuming the optimal online strategy (GoBack in this case), is around $1,333$ I/Os (we need to recompute the $80,000$ tuples; with a selectivity of 0.6, we need to read around $133,333$ R tuples). On the other hand, the overhead of SMJ would be (in the worst case, assuming the sort buffer is full) around 167 I/Os. The cost of the NLJ plan is now $11,333$ I/Os whereas the cost of SMJ is $10,267$ I/Os, which means that SMJ is now better. In fact, for any suspend point beyond $16,020$ tuples in the NLJ buffer, SMJ is expected to outperform NLJ. On average, suspends may occur halfway through the buffer; therefore, SMJ is better than NLJ on the average.

8 Related Work

Our techniques exploit the internal semantics of individual operators to support efficient suspend/resume of complex query plans in their entirety. To the best of our knowledge, we know of no published work that addresses the same problem at such a level.

Query Reoptimization Researchers have argued [13, 17] for iterating between optimization and execution for large queries. Our work is not intended to handle re-optimization after resume, so in general it cannot be used for switching between different iterations in that setting. Conversely, while it may appear that plan-switching techniques designed in that setting can be applied in query suspend/resume, they are generally inadequate for various reasons: (1) They may be designed for switching execution strategies of individual operators, and do not work on suspending an entire query. (2) They may still leave large amounts of state in memory across switching points, which is not an option for suspend/resume. (3) They may be designed to switch at particular

points in execution, and inefficient for unexpected suspends. Overall, we see our work as complementary and orthogonal. Nevertheless, several connections are worth pointing out.

Avnur et al. [1] propose adaptive reoptimization by switching join orders at *moments of symmetry*. While these moments are analogous to our operator checkpoints, our focus is on suspending and resuming and entire query rather than making local plan changes.

Markl et al. [17] propose progressive reoptimization of queries using a checkpoint operator that validates the optimality of the query plan. A reoptimization usually triggers re-execution from scratch or using intermediate results. On the other hand, we define checkpoints at minimal-heap-state points in physical operators, support suspend and resume at a finer granularity, including regular plans produced by standard query optimizers.

Shah et al. [20] address the problem of partitioning stateful data flow operators across shared-nothing nodes. Our work can handle the different problem of migrating an entire data flow to a different node, which poses challenges and opportunities not present in operator-level migration.

Ng et al. [18] propose dynamically reconfiguring query execution plans in a distributed environment. Ives et al. [11] propose dividing source data into regions with different plans. Our suspend and resume techniques can be incorporated into these strategies to provide suspend and resume support at the level of subplans.

Other Related Work The Condor DAGMan [12] tool addresses failure recovery for an application consisting of set of tasks with dependencies. While it prevents application restart, it does not support resumption at an intra-task level. Our physical approach goes further and makes suspend decisions on a per-operator basis.

Labio et al. [14] considered the problem of resuming interrupted data warehouse loads. Their load operators are black boxes, but can specify a set of high-level properties, which the system uses to optimize resume. Since we are dealing with database queries instead of black-box transforms, we go much further. Our approach exposes more optimization opportunities and challenges, and leads to more efficient resume. Furthermore, they do not have a smart suspend phase like ours, which can explore the trade-offs among various suspend/resume strategies using constrained optimization.

Liu et al. [15] target query execution in a memory-constrained environment where stateful operators may need to spill some state to disk. They address the problem of selecting the state to move to disk, to minimize performance hit. Our problem of query suspend/resume is different. We cannot retain any state in memory across suspend and resume, so we focus on a different trade-off: between dumping to disk and reconstructing from a checkpoint.

9 Conclusion

Database queries are often long-running and occupy a lot of resources, such as memory. A number of applications can benefit from database support for query suspend and resume on demand. This task is challenging especially on modern systems, where query operators often use large amounts of internal state during execution.

We proposed a novel query lifecycle that supports efficient suspend and resume of queries. Our approach exploits the semantics of individual operators to checkpoint proactively or reactively. We developed a suite of techniques and data structures for coordinating these asynchronous checkpoints and enabling suspend/resume, with minimal impact to normal execution. We also formulated and solved the problem of choosing the optimal suspend strategy at suspend time, using runtime statistics.

Experiments with both simple and complex query plans using PREDATOR showed that our techniques offer much lower total overhead while meeting suspend time constraints. Hybrid suspend strategies were found to often perform better than purist techniques where all operators choose the same strategy. The suspend-time optimizer incurred negligible overhead and was able to adapt dynamically to skewed distributions in input data, and outperform simpler optimizers based on table-level statistics.

10 References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*, 2000.

[2] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up Persistent Objects. In *SIGMOD*, 1994.

[3] B. Chandramouli, C. Bond, S. Babu, and J. Yang. On suspending and resuming queries. Technical report, Duke University, July 2006. `http://www.cs.duke.edu/dbgroup/papers/2006-cbby-qresume.pdf`.

[4] Condor Project. `http://www.cs.wisc.edu/condor/`.

[5] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[6] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[7] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi. Minimizing Completion Time of a Program by Checkpointing and Rejuvenation. In *SIGMETRICS*, 1996.

[8] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, 1990.

[9] G. Graefe. Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys*, 1993.

[10] IBM DB2 Query Patroller. `http://www.ibm.com/software/data/db2/querypatroller/`.

[11] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to Source Properties in Processing Data Integration Queries. In *SIGMOD*, 2004.

[12] J. Frey. Condor DAGMan: Handling Inter-Job Dependencies. `http://cs.wisc.edu/condor/dagman/`.

[13] N. Kabra and D. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD*, 1998.

[14] W. J. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient Resumption of Interrupted Warehouse Loads. In *SIGMOD*, 2000.

[15] B. Liu, Y. Zhu, and E. A. Rundensteiner. Run-Time Operator State Spilling for Memory Intensive Long-Running Queries. In *SIGMOD*, 2006.

[16] G. Luo, J. F. Naughton, C. Ellmann, and M. Watzke. Toward a Progress Indicator for Database Queries. In *SIGMOD*, 2004.

[17] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing Through Progressive Optimization. In *SIGMOD*, 2004.

[18] K. Ng, Z. Wang, R. Muntz, and E. Shek. On Reconfiguring Query Execution Plans in Distributed Object- Relational DBMS. In *ICPADS*, 1998.

[19] P. Seshadri. PREDATOR: A Resource for Database Research. In *SIGMOD Record*, 1998.

[20] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, 2003.