Cache-Oblivious Scheduling of Shared Workloads

Arian Bär #, Lukasz Golab *, Stefan Ruehrup #, Mirko Schiavone # and Pedro Casas #

# *Telecommunications Research Center Vienna (FTW), Vienna, Austria*
{baer, ruehrup, schiavone, casas}@ftw.at

* *University of Waterloo, Waterloo, Canada*
lgolab@uwaterloo.ca

***Abstract*—Shared workload optimization is feasible if the set of tasks to be executed is known in advance, as is the case in updating a set of materialized views or executing an extract-transform-load workflow. In this paper, we consider data-intensive workloads with precedence constraints arising from data dependencies. While there has been previous work on identifying common subexpressions and task re-ordering to enable shared scans, in this paper we solve the problem of scheduling shared data-intensive workloads in a cache-oblivious way. Our solution relies on a novel formulation of precedence constrained scheduling with the additional constraint that once a data item is in the cache, all tasks that require this item should execute as soon as possible thereafter. We give an optimal algorithm using A* search over the space of possible orderings, and we propose efficient and effective heuristics that obtain nearly-optimal schedules in much less time. We present experimental results on real-life data warehouse workloads and the TCP-DS benchmark to validate our claims.**

I. Introduction

There are several data management scenarios in which the workload consists of a set of tasks that are known in advance. For example, extract-transform-load (ETL) processing involves executing a predefined workflow of operations that pre-process data before inserting it into the database. Another example is data stream processing and publish-subscribe systems, in which a predefined set of queries is continuously executed on incoming data. Also, in data warehouses, materialized view maintenance is often done periodically, in which all the views, which are known in advance, are updated together.

Previous work has recognized optimization opportunities in these scenarios, referred to as shared workloads, including scan sharing, shared query plans and evaluating common sub-expressions only once [24]. In this paper, we address the following problem: given a shared workload, even after identifying common sub-expressions and shared scan opportunities, it is still not clear *what is an optimal ordering of tasks that minimizes cache misses?* Furthermore, since we may not know the exact amount of cache that is available to the data management system at a given time, we want to generate a task ordering in a *cache-oblivious way*, i.e., in a way that exploits caching without knowing the cache size.

Throughout this paper, we will use the term "cached results" in a general sense. Depending on the application, this could refer to the disk-RAM hierarchy or the RAM-cache hierarchy.

A. Motivating Example

While the solution presented in this paper is applicable to any data-intensive shared workload (i.e., where data I/O is the bottleneck, not CPU), our motivation for studying cache-oblivious task ordering comes from Data Stream Warehouses (DSWs) such as DataDepot [15] and DBStream [6]. DSWs are a combination of traditional data warehouse systems and stream engines. They support very large fact tables, materialized view hierarchies and complex analytics; however, in contrast to traditional data warehouses that are usually updated once a day or once a week, DSWs are refreshed more often (e.g, every 5 minutes) to enable queries over nearly-real time and historical data. Example applications include network, datacenter or infrastructure monitoring, data analysis for intelligent transportation systems and smart grid monitoring.

Since the "claim to fame" of DSW systems is their ability to ingest new data and refresh materialized views frequently, view maintenance must be performed efficiently. The system must finish propagating one batch of new data throughout the view hierarchy before the next batch arrives. Otherwise, at best a backlog of buffered data will build up, and at worst some data will be lost and not available for future analysis.

For example, consider the simple view hierarchy shown in Fig. 1, with 0 and 1 being base tables and the other nodes corresponding to materialized views. Note that some views (e.g., 2 and 4) are computed directly over base tables while others are computed over other views (e.g., 5). This is the predefined workload that a DSW will repeatedly execute when a batch of new data arrives for tables 0 and 1.

The view hierarchy forms a precedence graph. When a batch of new data arrives for table 1, we first insert it into table 1, and then we can use it to update views 2 and 4. Since view 5 needs view 2 as an input, it can only start processing after view 2 was updated. Thus, a legal ordering of view updates must satisfy the given precedence constraints; e.g., we cannot update view 5 if we have not yet updated view 2.

However, different legal orderings may lead to different cache performance. For example, right after updating table 1 with new data, that new batch of data is likely to be in the cache. Therefore, we should then update views 2 and 4 while the new batch of data is in the cache. On the other hand, if we update table 0 in between views 2 and 4, then the new batch of data from table 1 is more likely to be evicted and will have to be reloaded before updating view 4. Put another way, we would need a larger cache to avoid cache misses.

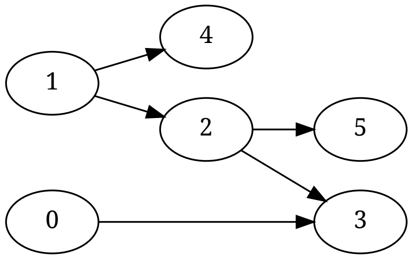


Fig. 1. A precedence graph corresponding to two base tables and four materialized views.

B. Challenges and Contributions

Even in the simple example above, it is not obvious which ordering minimizes cache misses. It makes sense to update views 2 and 4 immediately after updating table 1, but should we update view 2 before 4 or vice versa? As the number of tasks in the workload and their data dependencies increase, so does the complexity of choosing an efficient ordering. Furthermore, in practice we usually do not know exactly how much cache is available for a given task at a given time. For instance, in a DSW, view updates compete for resources with ad-hoc queries.

The intuition behind our solution is simple: tasks that require some data item should be scheduled as soon as possible after this item is placed in the cache. Otherwise, other tasks that require other data items will be scheduled, increasing the odds that the original data item will be evicted from the cache. In other words, we need to minimize the amount of time a data item (e.g., a new batch of data loaded into a materialized view) spends in the cache until all the subsequent tasks that need it have been executed.

For example, Fig. 2 illustrates two possible legal orderings of the five tasks from Fig. 1, obtained by linearizing the view precedence graph. For each node that includes at least one outgoing edge (e.g., each task that produces data required by other task(s)), we can compute how long these data must remain in the cache. At the top of the figure, the "distance" between table 0 and view 3, which requires data from table 0, is four, i.e., three other tasks will run in between. For view 2, the maximum distance is three, since both view 3 and view 5 need data from view 2, and a total of three view updates will run from the time view 2 data are inserted into the cache until both view 3 and view 5 updates are completed. On the other hand, the ordering shown at the bottom of the figure has a distance of only one between table 0 and view 3—they are executed one after the other and data from table 0 is more likely to still be in the cache at the time of execution of view 3. The idea behind our approach is to minimize the distance between related tasks and therefore decrease the possibility of cache misses, without having to know the cache size (our notion of distance will be formalized in Section II).

The specific contributions of this paper are as follows.

1) We formalize the problem of ordering tasks within shared data-intensive workloads to optimize cache usage, but without knowing the cache size. We formalize our objective of minimizing the distances between related tasks as minimizing the Total Maximum Bandwidth (TMB) of the resulting schedule, which extends two related classical problems: directed bandwidth and directed optimal linear arrangement [8], [13] (details in Section II).
2) We give an algorithm for finding an optimal ordering that uses A* search to efficiently examine the space of all possible orderings.
3) Since the optimal A*-based algorithm is infeasible in practice for all but the simplest workloads, we propose two heuristics that search a small subspace of possible orderings and return good orderings in much less time.
4) We experimentally show that the proposed heuristics obtain nearly-optimal answers on small problem instances, where it is feasible to compute an optimal solution, and we show the effectiveness and efficiency advantages of the heuristics against a baseline algorithm using real data stream warehouse workloads and the TPC-DS decision support benchmark.

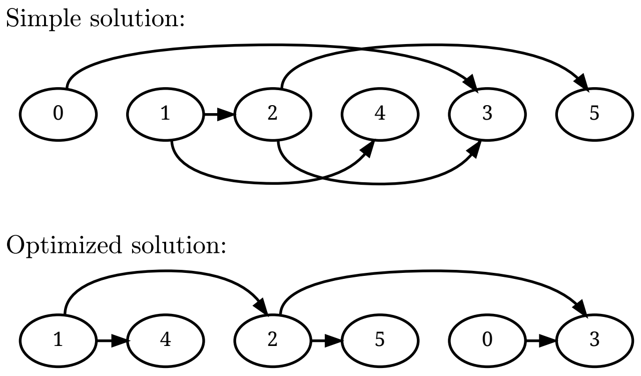


Fig. 2. A simple and a optimized ordering of the tasks from Figure 1.

C. Roadmap

The remainder of this paper is organized as follows. Section II formally defines our problem; Section III discusses related work in shared workload optimization, scheduling and cache-oblivious algorithms; Section IV presents the proposed algorithms; Section V discusses experimental results; and Section VI concludes the paper.

II. Problem Statement

The general problem we investigate in this paper is the scheduling of tasks with precedence constraints corresponding to data dependencies. Precedence constraints impose a partial order on the tasks. This partial order is given as input in the form of a directed acyclic graph (DAG) $G = (V, E)$, where each node $v \in V$ represent a task and each directed edge $e = (u, v) \in E$ is a precedence constraint, which requires that task u has to be scheduled before task v. Optionally, the input may include the size the output of each task; we will deal with this later on in this section. In addition to satisfying the given precedence constraints, we will impose optimization goals on the generated ordering to minimize cache misses.

G may consist of a number of connected components, e.g., a view hierarchy that uses some set of base tables, and another view hierarchy that is sourced from a different set of base tables. Since inter-dependencies and therefore cache optimization opportunities only exist in each connected component, we can deal with each connected component separately, and thus we assume from now on that G is connected.

We assume the tasks are data-intensive. That is, the bottleneck is loading the data into the cache rather than the subsequent processing; otherwise, even having an unlimited cache would not help much. We assume a *cache-oblivious* setting, in which we do not know the size or granularity of the cache. Further, we assume that the tasks belonging to a given connected component are to be scheduled serially on a single machine, although different connected components can be scheduled in parallel. We defer a full treatment of multi-threaded scheduling in our context, as well as handling task priorities, to future work.

Let $\sigma : V \rightarrow \{0, 1, ..., |V|\}$ be a schedule function that orders the tasks (i.e., the nodes in the precedence graph) in a given workload. The *precedence constrained scheduling* problem as formulated in [13] asks whether a schedule can meet a deadline. On a single-processor system, the problem of scheduling tasks with precedence constrains, without taking caching into account, is solvable in polynomial time [13]. However, real systems benefit from caching: the result of a preceding task u can be retrieved from the cache for task v, if the cache has enough capacity to keep the result of v despite other tasks that are scheduled between u and v. Therefore, minimizing the distance between u and v in the schedule σ, which is expressed by $|\sigma(u) - \sigma(v)|$, increases the likelihood of a cache hit.

There are two classical problems that express related objectives: (1) directed bandwidth, which aims to construct a schedule with a bound on the maximum distance of an edge in the precedence graph, and (2) directed optimal linear arrangement, which aims to construct a schedule with a bound on the sum of the distances for all edges:

Directed bandwidth (DBW) (GT41 in [13], GT43 in [8]): Given a graph $G = (V, E)$ and a positive integer K, is there a schedule function $\sigma : V \rightarrow \{1, ..., |V|\}$ such that $\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\max |\sigma(v) - \sigma(u)| \leq K\,? \tag{1}$$

Directed optimal linear arrangement (DLA) (GT42 in [13], cf. GT44 in [8]): Given a graph $G = (V, E)$ and a positive integer K, is there a schedule function $\sigma : V \rightarrow \{1, ..., |V|\}$ such that $\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\sum_{(u,v) \in E} |\sigma(v) - \sigma(u)| \leq K\,? \tag{2}$$

Both of the above problems are NP-complete [14], [23]. Note that the problems are defined as decision problems, for which the corresponding optimization problems can be shown to be equally complex.

In our context, solving the DBW problem only optimizes for the single longest edge in the entire workload and does not take any other data dependencies into account. DLA is not suitable in the context of caching as it was originally meant for scheduling of production workloads, in which a task produces multiple items, one for each subsequent tasks it is connected to. For example, recall Fig. 2 and note the edges from task 2 to tasks 3 and 5. DLA counts both of these edges, effectively assuming that two copies of the output of task 2 need to be stored. However, *we are only interested in the longest edge from a task to any subsequent task that depends on it*, as that determines how long the data generated by the initial task need to stay in the cache.

Based on the above observations, we formulate a new problem, *total maximum bandwidth*, that reflects our objective, which is a combination of DBW and DLA:

Total Maximum Bandwidth (TMB): Given a graph $G = (V, E)$ and a positive integer K, is there a schedule function $\sigma : V \rightarrow \{1, ..., |V|\}$ such that $\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\sum_{u \in V} \max_{\{v | (u,v) \in E\}} |\sigma(v) - \sigma(u)| \leq K\,? \tag{3}$$

If the input also includes the size of the output of each task u, call it ω_u, then we can extend TMB to *Weighted Total Maximum Bandwidth* (WTMB) by optimizing for the weighted distance of the longest edge from any task to a dependent task. For WTMB, the optimization problem becomes:

$$\sum_{u \in V} \omega_u \max_{\{v | (u,v) \in E\}} |\sigma(v) - \sigma(u)| \leq K\,? \tag{4}$$

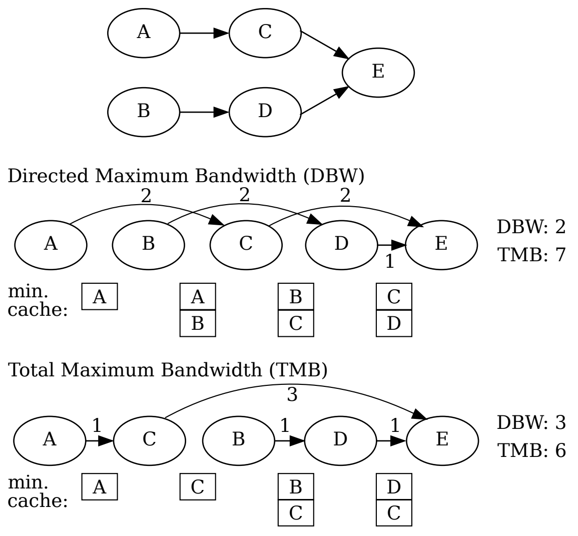
Examples

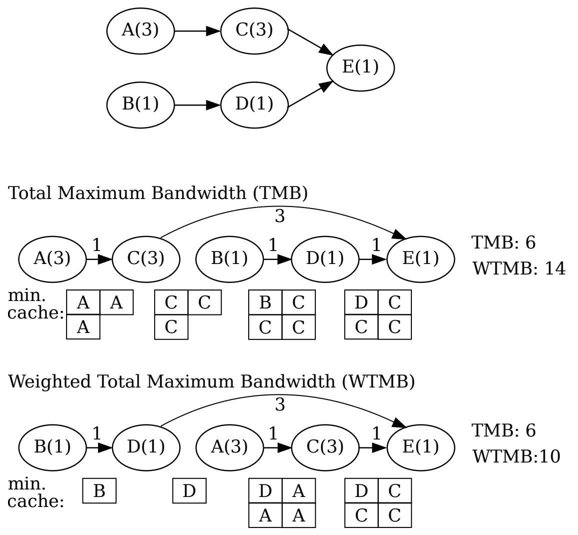
We close this section with two examples, one comparing the TMB and DBW cost functions and the other comparing TMB with WTMB.

Fig. 3 shows a precedence graph for five tasks (top), followed by two possible schedules (bottom). At the bottom of each schedule, we show the minimal cache contents at every step to avoid cache misses (labeled min. cache). The first schedule has an optimal DBW cost of only 2, the length of the longest edge. Its TMB cost is seven, which is the distance between A and C (of 2) plus the distance between B and D (of 2) plus the distance between C and E (of 2) plus the distance between D and E (of 1). To understand the cache contents illustrated below the schedule, note that when A is done, its output must be stored for C to use later. Then, when B is done, its output is stored for use by D. When C is done, its output is stored for use by E, but the output of A is no longer needed. When D is done, its output and C's output are stored for use by E, and B's output is no longer needed.

The lower schedule in Fig.3 has an optimal TMB cost of six, but its DBW cost is higher than that of the first schedule—here, the longest edge has a length of three. This schedule also requires less cache over time to avoid cache misses since only one item (namely C) has to be stored in the second step.

Fig. 4 compares two schedules for the same precedence graph as in Fig. 3 in terms of TMB and WTMB costs. The notation A(3) indicates that that size of the output of A is three. Both schedules have the same TMB cost of six, but the second one has a lower WTMB cost of ten (which is the distance between B and D of one, plus the distance between D and E of three, plus the distance between A and C multiplied by the size of A of three, plus the distance between C and E multiplied by


A
C
E
B
D
Directed Maximum Bandwidth (DBW)
2
2
2
1
DBW: 2
TMB: 7
min. cache:
Total Maximum Bandwidth (TMB)
3
1
1
1
DBW: 3
TMB: 6
Fig. 3. An example comparing DBW with TMB.


A(3)
C(3)
E(1)
B(1)
D(1)
Total Maximum Bandwidth (TMB)
TMB: 6
WTMB: 14
min. cache:
Weighted Total Maximum Bandwidth (WTMB)
TMB: 6
WTMB:10
Fig. 4. An example comparing TMB with WTMB assuming we know the size of the output of each task.

the size of E of three). The minimal cache contents are shown at the bottom, as before; note that the output of A and C now has size three and therefore both take up three cache slots, as illustrated. The bottom schedule has a lower WTMB cost, and requires less cache over time to avoid cache misses.

III. Related Work

This paper is related to three areas: cache-oblivious algorithms, optimizing shared workloads and scheduling.

Cache-oblivious algorithms have been studied for over a decade, beginning with Frigo et al. [10]. The idea is to make (nearly) optimal use of the cache without knowing its size, and a common approach has been to divide-and-conquer a given problem so that at some level the resulting sub-problems are small enough to fit in the cache, regardless of the size of the cache. There has been some very recent work on cache-oblivious scheduling [1], but only for the special case of a chain of streaming operators, which is not applicable to our problem of scheduling a DAG of operators.

In shared workload optimization, there has been work on efficiently refreshing a data warehouse consisting of a collection of materialized views. One set of optimizations focuses on sharing work among similar views; examples include finding common sub-expressions among similar views [19], [21], and choosing an optimal view graph when there are many possible source views to choose from [9]. There is also similar work in shared data processing systems, e.g., computing a global query plan [5], [12], sharing work across MapReduce jobs [22] and enabling shared table scans [11], [20], [27]. In this paper, we address a complementary issue: once a shared query plan is found, we order the execution of the tasks to maximize the chances of reusing cached results.

The other set of optimizations addresses ordering of view updates, and, to the best of our knowledge, there is no prior work on cache-oblivious ordering. Labio et al. considered select-project-join views and ordered the updates according to whichever view has the smallest delta [18]. Golab et al. proposed a scheduling algorithm for view refresh in an on-line data warehouse, which orders jobs by their improvement in freshness divided by the processing time [16]. This algorithm is for on-line settings where new data arrive asynchronously, and therefore work sharing may not be feasible.

Similarly, there has been work on query re-ordering to maximize sharing opportunities. Agrawal et al. addressed the problem of scheduling file scans given an expected frequency of queries that access each file [2]. Ahmad et al. [3] studied query re-ordering to take advantage of positive interactions among queries. Wolf et al. [26] deal with scheduling MapReduce jobs to enable shared scans. In our solution, we also reorder tasks to take advantage of sharing, but in a cache-oblivious way. Gupta et al. studied the problem of choosing which common sub-expressions to materialize in order to speed up the evaluation of a sequence of queries, given a fixed-sized cache to store the subexpressions [17]. Our solution does not require the knowledge of the cache size.

Finally, from a scheduling point of view, as we mentioned earlier, previous work on precedence constrained scheduling [7], directed optimal linear arrangement and directed bandwidth [8] cannot model our objective of minimizing the distance between related tasks that require the same data item(s). Scheduling with sequence-dependent setup times [4] is also related; in this problem, the execution time of each task includes some setup time that depends on the all the tasks that have been executed up to now, plus the actual task processing time. However, this work mostly considers production scheduling where physical objects are involved, leading to different assumptions and objectives.

IV. Algorithms

In this section, we present algorithms that take in a precedence graph and output a schedule optimized for TMB or WTBM if the task output sizes are known (Equations 3 and 4, respectively). We start by defining the concepts and subroutines that will be used by the algorithms (Section IV-A). We then present an optimal algorithm based on A*-search of the complete space of possible schedules (Section IV-B),

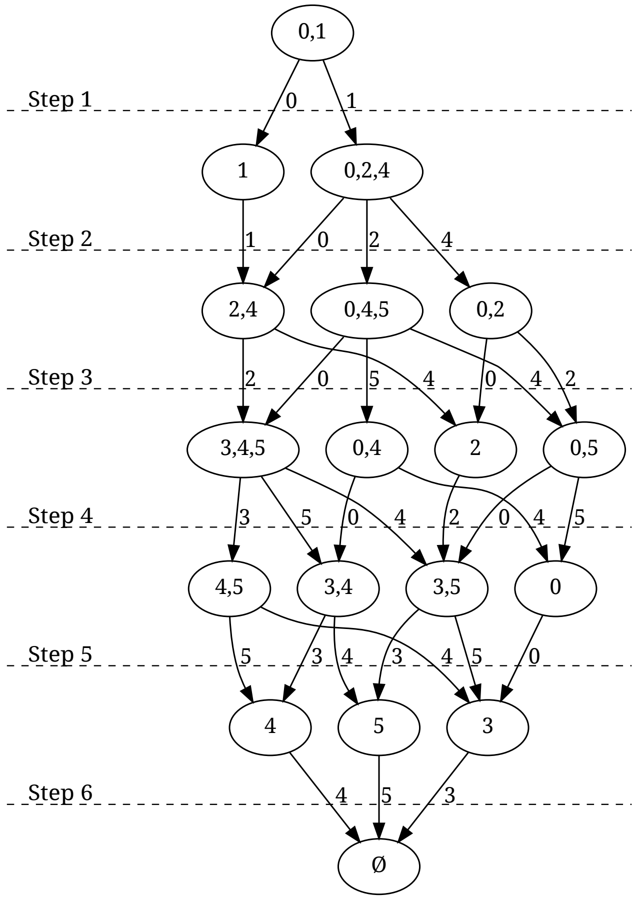


Fig. 5. Candidate search graph for the workload whose precedence graph was shown in Figure 1.

followed by three approximate algorithms that examine a subset of possible schedules: a simple breadth-first baseline approach (Section IV-C) and two heuristics, a greedy algorithm that always chooses a task whose distance to its predecessor is the smallest (Section IV-D) and an algorithm that chooses tasks which are likely to lead to efficient schedules (Section IV-E).

A. Preliminaries

First, we define the *candidate search graph*, $\overline{G} = (\overline{V}, \overline{E})$, in which the sequence of edge labels along every path from the start to the sink is a feasible schedule that obeys the precedence constraints encoded by the given precedence graph $G = (V, E)$[1]. For example, the candidate search graph corresponding to the precedence graph from Fig. 1 is shown in Fig. 5. Each node $\overline{v} \in \overline{V}$ denotes the schedulable tasks at that point in the schedule, i.e., those which can now be executed because all of their precedence constraints have been met. Each edge $(\overline{u}, \overline{v}) \in \overline{E}$ is labeled with the name of the task that is to be executed at that step. The start node at the top of Fig. 5 contains tasks 0 and 1, which must run before any other tasks. If we follow the right edge, labeled 1, the schedulable tasks are now 0, 2 and 4, and so on.

[1]A similar search graph was used in [25] in the context of the direct optimal linear arrangement problem.

We can construct $\overline{G}$ from G in a straightforward way. In the first step, the source node in $\overline{G}$ contains all the root nodes in G (i.e., the tasks without any predecessors). In each subsequent step, we create edges for all tasks contained in the labels of nodes created in the previous step, labeled with the task number. For each of the created edges, we created a new node in $\overline{G}$ that contains the tasks that are now schedulable (if such a node has not already been created). Finally, if no more nodes are schedulable, the edges are connected to the sink node, labeled with $\emptyset$.

Since we create $\overline{G}$ on-the-fly, the following definitions are based on a partial schedule. Let s be a possibly partial schedule of $|s|$ tasks. Let $get\_cands(G, s) := \{v : (u, v) \in E \text{ and } u \in s \text{ and } v \notin s\}$. That is, get_cands returns the set of schedulable tasks that can now be appended to s assuming that all the tasks in s have already been executed. Furthermore, for a task u, let $successors(u)$ be the set of tasks that depend on u, i.e., $successors(u) = \{v : (u, v) \in E(G)\}$.

Finally, we define a $tmb\_cost(s, G)$ function that evaluates the Total Maximum Bandwidth cost of a possibly partial schedule s according to Equation 3 (or Equation 4 if the task output sizes $\omega(u)$ are known). Let σ be the ordering function of s (recall Section II). For each task u in s, we compute $tmb\_cost(s, G)$ as follows.

1) If u has no successors, do nothing;
2) else if all of u's successors are already in s, add to the total cost the distance between u's last successor and u, i.e., $\max_{v \in successors(u)} |\sigma(v) - \sigma(u)|$, multiplied by $\omega(u)$ if given;
3) else (if not all of u's successors are in s), add to the total cost the quantity $|s| + 1 - \sigma(u)$, which is a lower bound on the distance between u's last successor (which has not yet been scheduled) and u (again, multiplied by $\omega(u)$ if given).

For example, consider the partial schedule $s = \langle 0, 1, 2, 4 \rangle$ for the precedence graph from Fig. 1. For task zero, the cost is four since its successor, task 3, appears four positions later in the sequence. For task 1, the cost is two since its successor, task 4, appears two positions later. The cost for task 2 is two, which is a lower bound for its true cost (its successor, task three, has not yet been scheduled). Finally, the cost for task 4 is zero since it has no successors. Thus, $tmb\_cost(s, G) = 4+2+2+0 = 8$.

Algorithm 1 shows the pseudocode for the WTMB cost function. Each task in the possibly partial schedule s is considered sequentially. Lines 6 through 11 count how many of the current task's successors are in s and record the position in s of the furthest successor of the current task. Note the use of the position function σ to find the position of outTask in s. Line 12 counts the total number of successors of a given task. If this number is zero, the current task does not incur any TMB cost. Otherwise, if all the successors have already been scheduled, we can precisely compute the TMB cost in line 16, which is simply the difference in the position of the furthest successor and the task itself. Otherwise, line 18 computes a lower bound on the given task's TMB cost. In line 20, we add the cost of the current task to the total cost of the schedule. Note the ω function that determines the output sizes for WTMB. In case of TMB, $\omega(task)$ is simply one for every task.

Algorithm 1 (Weighted) tmb_cost

```
cost = 0  // the overall cost of the schedule
for task in s do
    stepCost = 0
    maxOutPos = 0 // position of furthest successor
    outTasksDone = 0
    for outTask in successors(task) do
        if outTask in s then
            outTasksDone++
            maxOutPos = max($\sigma$[outTask], maxOutPos)
        end if
    end for
    $\ell$ = len(successors(task))
    if $\ell$ == 0 then
        do nothing // no successors
    else if outTasksDone == $\ell$ then
        stepCost = maxOutPos - $\sigma$(task)
    else
        stepCost = len(s) - $\sigma$(task)
    end if
    cost += $\omega$(task) * stepCost
end for
return cost
```

B. Optimal Algorithm Based on A* Search

We begin with an optimal algorithm based on A* search that considers every possible schedule and selects an optimal one with the lowest tmb_cost. As we will experimentally show in Section V, this algorithm is not feasible in practice for non-trivial problem instances because the number of possible schedules can be prohibitively large.

A* search finds a least-cost path between two nodes, in our case the start node and the sink node of the candidate search graph (i.e., a least-cost schedule). For each node x, the cost function used by A* includes two parts: $g(x)$, which is the cost of the path from the start node to x, and $h(x)$ which is a heuristic function that approximates, but must not overestimate, the cost of the path from x to the sink node. In our problem, $g(x)$ is simply $tmb\_cost(s)$ where s is the schedule corresponding to the path from the start node to x. The more interesting part is $h(x)$.

To solve our problem, we define $h(x)$ as the sum of the outgoing edges present in the precedence graph for each task that has not yet been scheduled along the path from the start node to x. To understand why this is an admissible function for A* search (i.e., one that does not overestimate the remaining cost of the path), note that if a task node has an outgoing edge in the precedence graph, then there is a successor task that must be scheduled after that node. Thus, a lower bound on the total maximum bandwidth cost for the given task is the number of its outgoing edges in the precedence graph, i.e., the number of its successors. This lower bound occurs if all the successors are scheduled immediately after the given task. If any other task is scheduled before the last successor, the cost can only increase.

For example, consider the partial schedule $s = \langle 0, 1, 4 \rangle$ based on Fig. 1. The $g(x)$ function of the node in the candidate search graph corresponding to this partial schedule is simply $tmb\_cost(s, G)$, which is 5 (three for task zero and two for task one, since not all of their successors have been scheduled, and zero for task 4 because it does not have any successors). To compute $h(x)$, note that tasks 2, 3 and 5 are yet to be scheduled. The sum of the outgoing edges of these three nodes in the given precedence graph is two, which gives us $h(x)$. Thus, the total cost of s as computed by A* search is $g(x) + h(x) = 7$. It is easy to verify that no complete schedule with s as its prefix can have a tmb_cost of less than 7.

C. Baseline Algorithm

We now present the first of three algorithms that consider a subset of the possible schedules and therefore are faster than the A*-based algorithm, but are not guaranteed to find a good solution. The first such algorithm is the simplest and fastest approach we refer to as *Baseline*: at every step, it randomly chooses one of the currently-schedulable tasks. Thus, using the precedence graph from Fig. 1 as input, in the first step, Baseline executes tasks 0 and 1 in random order, then tasks 2, 3 and 4 in random order, and then task 5. The running time of Baseline corresponds to that of breadth-first-search, which is $\mathcal{O}(|V| + |E|)$.

D. Greedy Algorithm

The next algorithm is the standard greedy heuristic applied to our problem: at every step, it chooses a schedulable task that yields the lowest $tmb\_cost(s, G)$ when added to the current partial schedule s. Ties are broken randomly.

Using the precedence graph from Fig. 1 as input, the greedy heuristic first decides between tasks zero and 1. For both s=$\langle 0 \rangle$ and s=$\langle 1 \rangle$, $tmb\_cost(s, G) = 1$ since not all of 0's or 1's successors, respectively, have been scheduled. Suppose the tie-break results in task 1 being sequenced first. In the next step, the schedulable tasks are still zero, plus 2 and 4. For s=$\langle 1, 0 \rangle$, $tmb\_cost(s, G) = 2$. For s=$\langle 1, 2 \rangle$, $tmb\_cost(s, G)$ is 2 due to task 1 plus 1 due to task 2, which gives 3. For s=$\langle 1, 4 \rangle$, $tmb\_cost(s, G) = 2$ due to task 1 (plus zero due to task 4 since it has no successors). Thus, the greedy algorithm randomly chooses between task 0 and 4 to follow task 1. We omit the remaining steps for brevity.

We now analyze the runtime complexity of the greedy algorithm. It uses the get_cands function to retrieve the set of currently schedulable tasks. However, since each step of the algorithm adds one task to the schedule, only the successors of this new task need to be added to the schedulable set. This gives $\mathcal{O}(|V| + |E|)$ for all get_cands calls over all the iterations. The runtime is dominated by calling tmb_cost for the considered schedules, which requires looping over all the outgoing edges of the tasks already in the schedule. This gives $\mathcal{O}(|E|)$ per call. Since the algorithm iterates $|V|$ times, and, clearly, at every iteration there are no more than $|V|$ schedulable tasks, for which tmb_cost is evaluated, the overall complexity of the greedy algorithm is $\mathcal{O}(|V|^2|E|)$.

E. Heuristic Algorithm

Our final algorithm is called *Heuristic*. In contrast to the greedy algorithm, which only examines the tmb_cost of adding every schedulable task to the current schedule in each iteration, the heuristic algorithm computes a complete feasible schedule for each schedulable task in every iteration, and chooses the task with the lowest-cost complete schedule. However, to keep the running time manageable, the heuristic algorithm cannot explore every possible feasible schedule (as does the A* algorithm). Instead, the complete schedules for each schedulable task are heuristically computed via deepest-first traversal, as explained below.

First, the heuristic algorithm pre-processes the precedence graph G by adding depth information to each node, corresponding to the distance to the furthest ancestor. For instance, in Fig. 1, the depth of tasks zero and 1 is zero, the depth of tasks 2 and 4 is one, the depth of task 5 is two, and the depth of task 3 is also two (its distance to task zero is one, but the distance to its other ancestor, task 1, is two).

Next, we illustrate what happens in the first iteration using Fig. 1 as input. Initially, the only schedulable tasks are zero and 1. We need to build complete schedules starting with zero and 1, respectively, compute their tmb_cost, and choose the task whose complete schedule has a lower tmb_cost (and we break ties arbitrarily).

The complete depth-first schedule that starts with task 1 is computed as follows. After task 1 has been scheduled, the schedulable tasks are zero, 2 and 4, of which either 2 or 4 have the largest depth. Let us assume task 2 is chosen next. The schedulable tasks then become zero, 4 and 5, of which is chosen because its depth is the largest. With the partial schedule now $\langle 1, 2, 5\rangle$, the schedulable tasks are zero and 4. We choose 4, and finally zero and three. This gives the complete schedule $s = \langle 1, 2, 5, 4, 0, 3\rangle$. Its tmb_cost is three for task 1, 4 for task 2, and one for task zero, which gives 8.

Similarly, the complete depth-first schedule that starts with task zero is computed as follows. After task zero has been scheduled, the only schedulable task is 1, so we choose it. Next, we have a choice between tasks 2 and 4, both of which have the same depth, so let us say we choose task 2. Then, the schedulable tasks are 3, 4 and 5, of which 3 and 5 have the highest depth, so let us say we choose task 3. This leaves tasks 4 and 5, and we choose 5 first because its depth is higher. This gives a complete schedule of $\langle 0, 1, 2, 3, 5, 4\rangle$, whose tmb_cost is three for task 0, 4 for task 1 and 2 for task 2, which is 9.

Thus, at the end of the first iteration, the Heuristic algorithm chooses task 1 and the second iteration begins. Note that the complete schedules calculated in the first iteration are now discarded and new complete schedules will be built in the second iteration, all of which will have task 1 scheduled first.

Fig. 6 summarizes the way in which the heuristic algorithm traverses the candidate search graph using Fig. 1 as input. As we described above, in the first iteration, two complete schedules are built, one starting with task zero and one starting with task 1. The latter is chosen by the heuristic algorithm, indicated by the bold arrow. The tmb_cost is also shown in the figure; note the cost of 9 if we choose task 0, versus the cost of 8 if we choose task 1. In the second iteration, the heuristic algorithm considers tasks 0, 2 and 4, and computes the corresponding three depth-first schedules. Choosing task 4 next is the best option. After task 4 has been selected, the algorithm computes two new depth-first complete schedules corresponding to adding tasks 0 and 2, respectively, to the existing partial schedule of $\langle 1, 4\rangle$. Adding task 2 is cheaper,

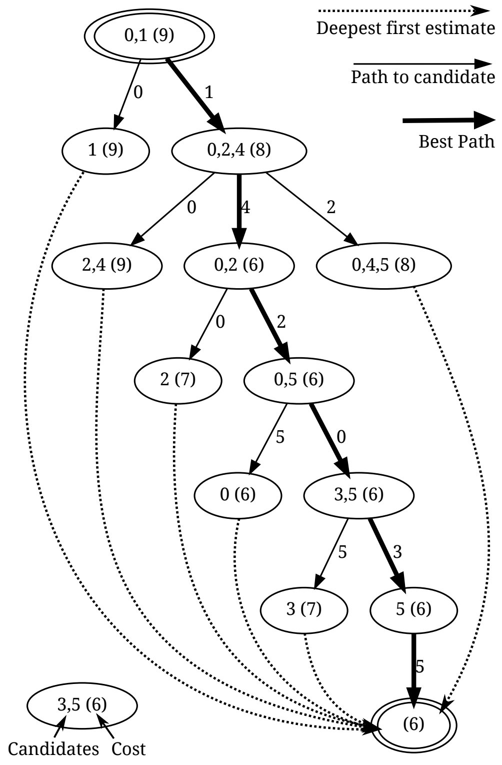


Fig. 6. Visualization of a run of the Heuristic algorithm on the candidate search graph created from the precedence DAG in Fig. 1.

as shown in the figure. The complete schedule generated by the heuristic algorithm is indicated by the bold arrows: $\langle 1, 4, 2, 0, 3, 5\rangle$. Its tmb_cost is 6.

The intuition behind computing complete schedules in a depth-first manner is to schedule successors right after their ancestors; notice that when a task with a higher depth than the previous task is chosen, these two tasks should be very close together in the topological sort of the precedence graph. However, any other heuristic for building possible complete schedules for a given schedule prefix is compatible with the framework we have described in this section.

Finally, we discuss the time complexity of the heuristic algorithm. Pre-processing the precedence graph to compute depth information (i.e. longest paths to a root in G) can be done via a linear-time shortest-paths algorithm on G with negative edge weights (this is only possible because G is a DAG, where after edge weight negation no negative cycles are possible). Now, one iteration of the algorithm involves computing multiple complete schedules in a deepest-first manner. For each such complete schedule, exactly one task is moved from the schedulable set to the actual schedule, and the successors of this task are added to the schedulable set. Therefore, each node and edge in G need to be visited only once. If the set of schedulable tasks is maintained in a data structure such as a binary heap that allows retrieval and deletion of the minimum-depth node and insertion in $\mathcal{O}(\log |V|)$, computing one complete schedule requires $\mathcal{O}(|E| + |V| \log |V|)$.

The overall heuristic algorithm iterates $|V|$ times. In each iteration, there are at most $|V|$ schedulable tasks, each of which requires a complete schedule to be built, at a cost of

$\mathcal{O}(|E| + |V|\log|V|)$, and its tmb_cost must be computed, but the former dominates the runtime. This gives the overall runtime complexity of the heuristic algorithm as $\mathcal{O}(|V| \cdot |V| \cdot (|E| + |V|\log|V|)) = \mathcal{O}(|E| \cdot |V|^2 + |V|^3 \log|V|)$.

V. Experimental Evaluation

This section presents our experimental findings on the effectiveness and efficiency (for both TMB and WTMB) of the algorithms we presented in Sec. IV (A*, Baseline, Greedy and Heuristic). We start with a description of our data sets and experimental environment (Sec. V-A), followed by the results:

- In Sec. V-B, we experiment with different precedence graphs as inputs, and we report the TMB/WTMB scores obtained by each algorithm as well as the time it took to generate the schedules. In general, we find that Heuristic obtains nearly-optimal schedules, but is slower than Greedy and Baseline (but still much faster than A*). Furthermore, both Greedy and Heuristic generate significantly better schedules than Baseline.
- In Sec. V-C, we run the proposed algorithms (except A*) on very large random precedence graphs to see if they can efficiently compute schedules for complex workloads. We found that Heuristic does not scale as well as Baseline and Greedy, but can still handle large precedence graphs.
- In Sec. V-D, we execute various workloads in PostgreSQL and show the real-world performance improvements due to our scheduling algorithms. Again, Heuristic and Greedy outperform Baseline.

A. Experimental Setup

We used a dual CPU Xeon E5-2630 machine, with 64 GB of RAM, and a 10-disk RAID10 storage subsystem. As a database we use PostgreSQL 9.2.4. We implemented all the algorithms presented in Sec. IV in the Go language (http://golang.org).

The `pgfincore` (http://pgfoundry.org/projects/pgfincore/) library is used to advise the operating system to drop tables from the disk cache. We use this functionality to evict tables from the cache when they are no longer needed. We will explicitly state whenever we make use of this function.

We used the following groups of data sets. The number of nodes and edges in the corresponding precedence graphs is shown in Table I, under the columns labeled $|V|$ and $|E|$. Below, we also provide the depth of each precedence graph, measured as the number of nodes on the longest path. Further details on the precedence graphs, as well as the source code of our algorithms and the random graph generator, may be found at https://github.com/arbaer/schedule.

- *running* corresponds to the running example from Fig. 1 with a depth of 3.
- *test1, test2, test3* and *test5* correspond to small hand-crafted workloads with various features. Their depths are three, four, five and four, respectively; however, *test5* additionally contains a node with a very high fan-out.
- *realworld1* and *realworld2* are two network monitoring workloads from data warehouses we are currently operating using the DBStream DSW [6]. The tasks are base tables and materialized view updates. *realworld1* has a depth of 5 and contains two nodes with a high fan-out, whereas *realworld2*'s depth is 6, but the fan-out is lower.
- *tpc-ds-scan, tpc-ds-7q, tpc-ds-11q* and *tpc-ds-63q* are based on the TPC-DS decision support benchmark (http://www.tpc.org/tpcds/). TPC-DS contains 24 base tables (7 fact tables and 17 dimension tables) and 99 predefined queries over the base tables. The number of tables required by a query ranges from one to 13, with an average of 4. We generated two versions of the benchmark: one with a scale factor of 10 and one with 100. All TPC-based precedence graphs have a depth of 2 since they consist of a layer of queries accessing a set of tables.
  The *tpc-ds-scan* workload consists of scan queries over the largest base tables: *catalog_sales, web_sales* and *store_sales*. These three tables account for 8.4GB in the scale-factor-10 version. The other three workloads contain 7, 11 and 63 queries from TPC-DS; the corresponding precedence graphs get progressively more complex. We do not use all 99 queries as not all of them are data-intensive and benefit from caching. The precedence graph for *tpc-ds-7q* is shown in Fig. 7.
- In one experiment, we also use large randomly-generated precedence graphs to test the scalability of our algorithms. These will be described later.

We use the small *running* and *test* workloads to test how close the solutions obtained by the heuristics are to an optimal solution; these are the only workloads on which it was feasible to run the A* algorithm. The *realworld* and TPC-DS workloads show that Greedy and Heuristic are scalable and outperform Baseline. We have the task output sizes for *realworld* and TPC-DS, so these are also used to test the WTMB version of our problem.

B. Comparison of Scheduling Algorithms

In the first set of experiments, we generate schedules using all the algorithms, and report how long it takes to create the schedules and the TMB cost of the schedules (even though we know the output sizes for some workloads, we ignore them for now and will consider WTMB shortly). We do not actually run the workloads in a database system. Table I shows the results. Each row corresponds to a different workload (we omit *tpc-ds-scan* as it is only relevant to the PostgreSQL experiments later in this section). For Baseline, Greedy and Heuristic, we report the mean TMB score and the standard deviation (SD) over 100 runs, since these algorithms break ties randomly and therefore may return different schedules for the same input. Bold TMB numbers indicate the best algorithms. For Heuristic and A*, we also count the number of node visits during execution (the number of node visits for Baseline and Greedy is small and not reported). Note that A* finished running within one hour only on small problem instances and the number of nodes it visits is very large.

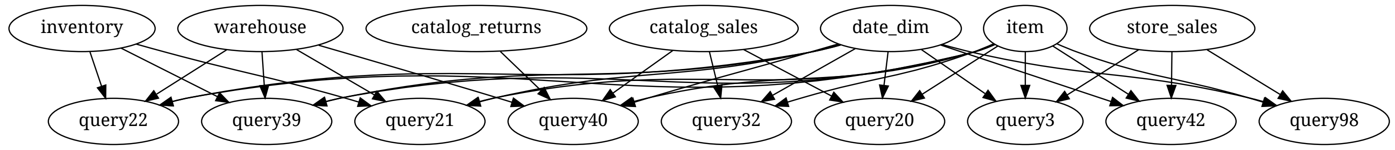


Fig. 7. Precedence graph for the 7 selected TPC-DS queries and tables in *tpc-ds-7q*.

TABLE I. PERFORMANCE COMPARISON OF THE IMPLEMENTED ALGORITHMS. TIMES MARKED WITH * INDICATE THAT THE EXPERIMENT WAS STOPPED AFTER 1 HOUR OF WALL CLOCK TIME.

			Baseline			Greedy			Heuristic				A*		
			TMB		time	TMB		time	TMB		time	nodes	TMB	time	nodes
Graph	$\|V\|$	$\|E\|$	Mean	SD		Mean	SD		Mean	SD			opt		
running	6	5	9	0.00	0.010s	8	0.00	0.012s	**6**	0.00	0.016s	18	**6**	0.019s	33
test1	5	4	6	0.00	0.010s	**5**	0.00	0.013s	**5**	0.00	0.012s	13	**5**	0.012s	21
test2	11	10	16.16	0.64	0.010s	16.16	1.32	0.011s	**14**	0.00	0.017s	34	**14**	0.031s	336
test3	22	21	65.54	3.69	0.010s	47.52	6.62	0.012s	37.44	0.50	0.038s	90	**35**	1.451s	12.4K
test5	25	30	91.58	12.96	0.010s	66.68	5.91	0.014s	48.40	0.49	0.053s	123	**46**	37.771s	325K
realworld1	43	48	269.1	11.8	0.048s	107.2	16.9	0.022s	**80.0**	1.8	0.075s	163	–	1h*	25.0M
realworld2	57	69	297.6	24.6	0.028s	120.4	10.3	0.023s	**119.2**	11.8	0.184s	616	–	1h*	29.5M
tpc-ds-7q	14	25	62.0	2.9	0.017s	52.8	3.7	0.019s	43.9	2.8	0.018s	55	**40**	0.27s	56.6K
tpc-ds-11q	31	70	362.4	13.3	0.028s	311.7	15.3	0.020s	**220.5**	11.8	0.054s	313	–	1h*	43.0M
tpc-ds-63q	85	310	1488.2	60.2	0.099s	1162.5	78.7	0.046s	**853.2**	12.2	1.505s	1598	–	1h*	16.2M

To summarize the results so far: for the workloads where A* was able to finish, we see that Heuristic gives a nearly-optimal schedule. Greedy also works well for some of the smaller problem instances. Baseline gives the most expensive schedules. On the other hand, Baseline and Greedy are extremely fast, Heuristic is still very fast, and A* is the slowest.

Fig. 8 compares the TMB and WTMB costs of the schedules returned by Heuristic, Greedy and Baseline for the workloads that come with output sizes, namely real-world and TPC-DS (these workloads are too large for A* to handle). The average costs and error bars are included, based on 100 runs of each algorithm.

Fig. 8a starts with the TMB costs that were already reported in Table I, indicating that both Greedy and Heuristic are significant improvements over Baseline, and that Heuristic is the overall winner (but it takes longer to compute the schedules).

In Fig. 8b, we show the WTMB costs of the schedules from Figure 8a, i.e., we have the algorithms optimize for TMB as before, but we compute the WTMB score of the resulting schedules by incorporating output sizes (which, of course, were not given to the algorithms). Heuristic continues to give the best and most stable results—note the wide error bars for Baseline and Greedy. In particular, different runs of Greedy may give widely different TMB results. For instance, if there are several base tables with various sizes but same TMB costs, Greedy randomly chooses the first table to update, regardless of the table size.

Note that the y-axis scales of Fig. 8a and Fig. 8b are different. TMB effectively assumes that each output size is one, whereas the WTMB scores are much higher because they reflect the true sizes of the inputs.

Fig. 8c shows the WTMB scores assuming the algorithms know the output sizes and are actually optimizing for WTMB, not TMB. Comparing to Fig. 8b, which has the same y-axis scale, knowing the output sizes clearly helps to lower the WTMB score of the resulting schedules for Greedy and Heuristic. Interestingly, Greedy slightly outperforms Heuristic in this experiment, meaning that greedily selecting tasks with the lowest WTMB scores is a good strategy (and there are no more ties that Greedy has to break randomly, unless the output sizes are exactly the same). We hypothesize that Heuristic could be tuned for the WTMB problem, e.g., by incorporating output sizes in the deepest-first schedule generation, but even now it is not much worse than Greedy.

TABLE II. SCALABILITY RESULTS.

Graph	$\|V\|$	$\|E\|$	Baseline	Greedy	Heuristic
rand1	100	200	0.02s	0.04s	0.84s
rand3	300	600	0.04s	0.95s	40.01s
rand5	500	1K	0.09s	4.59s	5m43s
rand6	1K	2K	0.34s	34.20s	85m46s
rand7	2K	4K	0.71s	4m22s	–
rand8	4K	8K	2.75s	31m47s	–

C. Scalability Comparison

In this experiment, we randomly generate very large precedence graphs (much larger than those used in the previous experiment) and measure the running time of Baseline, Greedy and Heuristic. Table II reports the number of nodes and edges for each random graph and the running times. Baseline and Greedy are very simple algorithms and scale extremely well. Heuristic does not scale as well, but can still handle graphs with up to 1000 nodes and edges in reasonable time (few or tens of minutes).

D. PostgreSQL Experiments

In this set of experiments, we execute various workloads under various schedules in the PostgreSQL database to measure the real-world performance improvements of our techniques. Here, we focus on the disk-RAM hierarchy.

Experiment 1: We start by running the simple *tpc-ds-scan* workload of three queries that scan three base TPC-DS tables:

```
Q1: select count(*) from catalog_sales;
Q2: select count(*) from web_sales;
Q3: select count(*) from store_sales;
```

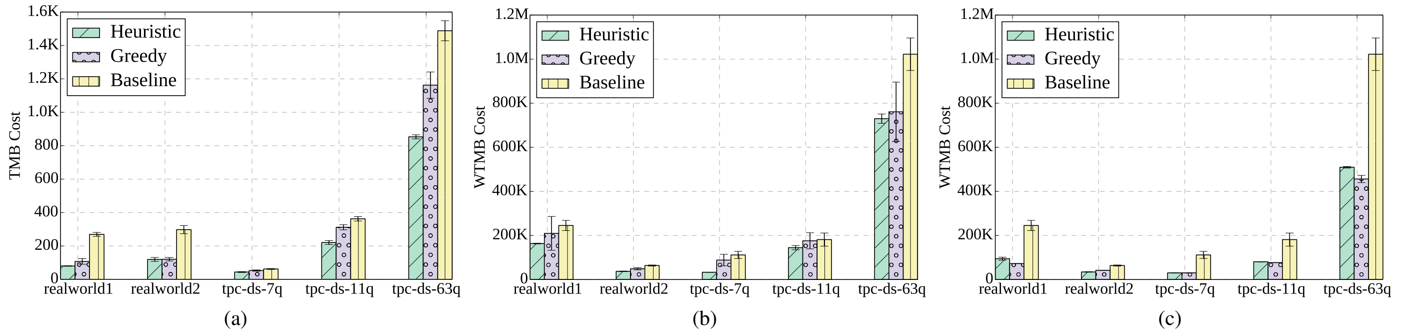


Fig. 8. Comparison of a) TMB costs, b) WTMB costs assuming the algorithms are optimizing for TMB, and c) WTMB costs assuming the algorithms are optimizing for WTMB.

In PostgreSQL, these queries result in full table scans, resulting in an I/O intensive workload. We also use the following three functions calls to drop tables from the cache:

```
X1: select drop_table_cache(catalog_sales);
X2: select drop_table_cache(catalog_sales);
X3: select drop_table_cache(catalog_sales);
```

We create three schedules, $S1$, $S2$ and $S3$, executing each query three times to demonstrate the differences in running time. In schedule $S3$, we also actively evict tables from the cache when they are not needed any more, indicated by the operations `X1`, `X2` and `X3`.

```
S1: Q1,Q2,Q3,    Q1,Q2,Q3,    Q1,Q2,Q3
S2: Q1,Q1,Q1,    Q2,Q2,Q2,    Q3,Q3,Q3
S3: Q1,Q1,Q1,X1, Q2,Q2,Q2,X2, Q3,Q3,Q3,X3
```

In Fig. 9a and 9b, we show the processing time and read I/O, respectively, of the three schedules under varying amounts of available cache (RAM), ranging from 500MB to 10GB in steps of 100MB. We control this by running a program that allocates and fills a specific amount of memory, making that amount unavailable to the database. In each experimental iteration, we first execute one schedule, force the operating system to drop all caches and then execute the next schedule.

Fig. 9a reports the processing times. *Schedule 1* is clearly the least efficient, so long as either no data fit into RAM (under 2GB) or all data fit into RAM (over 8.4GB). Additionally, *Schedule 3* achieves even better performance since tables that are not required any more are explicitly removed from the cache, simulating a optimal cache eviction strategy. The largest difference occurs at 4.5 GB of free RAM since now the biggest of the three tables fits entirely into the disk cache. At this point, *Schedule 1* finishes in 45 seconds while *Schedule 2* and *Schedule 3* in 33 seconds and 26 seconds, respectively. The resulting performance increase of *Schedule 3* over *Schedule 1* is 73 percent.

Fig. 9b illustrates the amount of disk read I/O during the run of this experiment under the same available cache conditions as before. These results indicate that there is a correlation between the amount of disk read I/O during the execution of a schedule and its execution time. *Schedule 1* needs to fit nearly all the tables into the cache before larger amounts of data can be reused. In *Schedule 2*, much more data can be reused through the cache and but even for larger amounts of available cache, between 4.5 and 8.4 GB, some data has to be fetched multiple times from disk. Finally, in *Schedule 3* since an optimal cache eviction strategy is applied, as soon as the biggest table fits entirely into the cache, data are only fetched once from disk.

We conclude from this experiment that changing the execution order of a workload can reduce the amount of disk I/O if not all the data fit into the cache, which also influences the execution time of a workload if it is I/O bound.

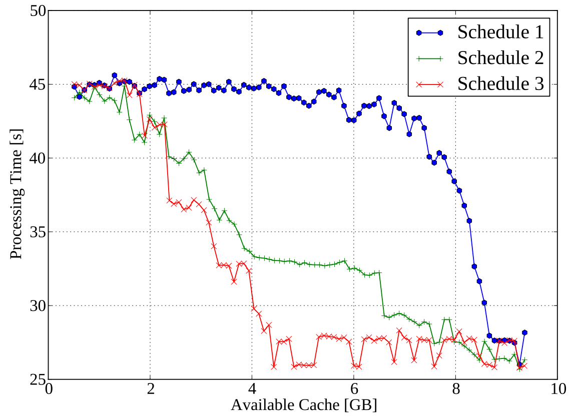
Experiment 2: Next, we show that the reduced amounts of disk read I/O are reproducible with queries from the TPC-DS benchmark. We use the *tpc-ds-7q* workload, consisting of 7 data-intensive queries, and execute them on TPC-DS tables generated using scale factor 100. This gives approximately 100GB of data. During the execution of a schedule, whenever a table is no longer needed in that schedule, it is evicted from the cache using the `drop_table_cache()` function, simulating a optimal cache eviction strategy.

We create schedules using Baseline, Greedy and Heuristic, each *not* considering the sizes of the outputs (tables), i.e., optimizing for TMB, not WTMB. We run the experiment four times and reduce the total amount of available system memory from 64 GB to 16 GB in steps of 16 GB, simulating machines with different amounts of available memory. We do this in the same manner as in the previous experiment.

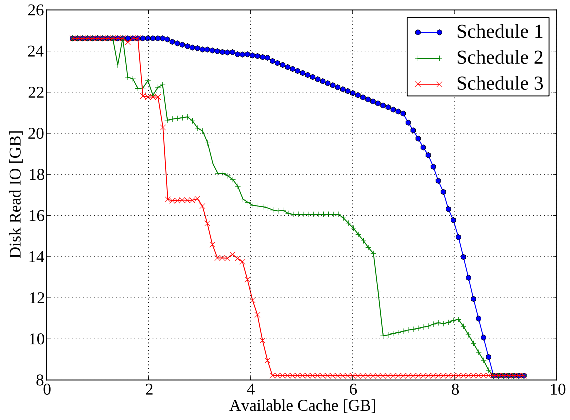
Fig. 10a shows the results, with disk read I/O on the y-axis. For 64 and 48 GB, only the Baseline schedule shows increased amounts of disk I/O. At 32 GB of available RAM, Heuristic performs better than Baseline. Although Greedy is also better than Baseline, it does not perform as well as Heuristic. Finally, for the 16 GB case all schedules perform nearly the same.

Fig. 10b shows the results of a similar experiment, but one in which we also give the algorithms the table sizes, allowing the algorithms to optimize for WTMB. The main difference compared to Fig. 10a are the reduced I/O reads for the 32 GB run for Greedy and Heuristic. Even in the 16 GB case, these algorithms perform better than Baseline.

If we extrapolate the memory needs linearly, the Greedy and Heuristic schedules would need about 320 GB to execute without additional read I/O for the larger TPC-DS scale factor of 1000.

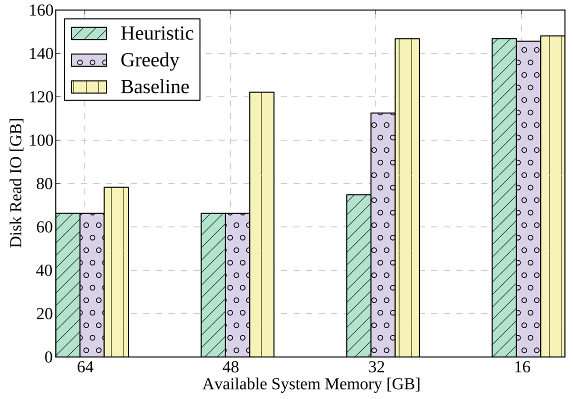


(a) Processing time

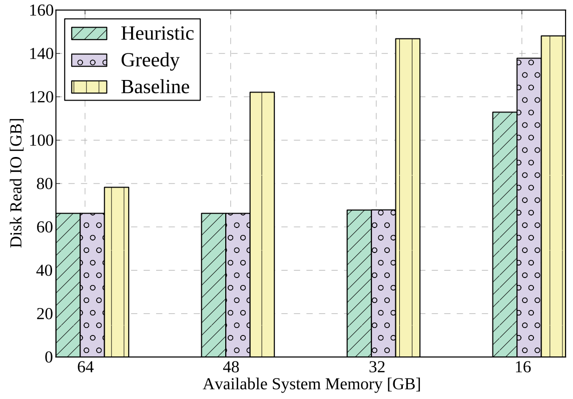


(b) Disk I/O

Fig. 9. Performance analysis executing the same workload with different schedules and increasing cache size.



(a) Algorithms **not** considering table sizes.



(b) Algorithms considering sizes.

Fig. 10. Disk read I/O for the `tpc-ds-7q` workload scheduled by the Heuristic and the Baseline algorithms.

Experiment 3: In the last experiment, we study the effect of different schedules on cache usage over time. We use the *tpc-ds-63q* workload, which contains the 63 most data-intensive queries. We use scale TPC-DS scale factor 10. Only the raw data are imported into the database, without creating any further auxiliary data structures such as indices. We do not optimize the queries in terms for processing time since our focus is on I/O optimization via scheduling to exploit the cache. Before each experimental run, we first clear the cache and then we execute the *tpc-ds-63q* workload in the order specified by the scheduling algorithm. Every second, we sample the disk cache usage. As soon as a table is no longer needed by any other query in the schedule, we remove it from the cache using the `drop_table_cache()` function. Note that we never actively load any data into the cache, but instead tables are loaded automatically when they are accessed by a query.

Fig. 11a shows the RAM usage (on the y-axis) as a function of time since the start of the experiment (x-axis) for Baseline, Greedy and Heuristic optimizing for TMB (i.e., the algorithms do not know the table sizes). Fig. 11b shows the results for WTMB. In both cases, the cache usage over time is greatly reduced by Greedy and Heuristic as compared to Baseline.

Finally, we compare the algorithms by their areas under the cache usage curves. If $c_s(t)$ denotes the cache usage of schedule s at any given point in time t, we can formulate the average cache usage for the total execution time ΔT as $\overline{cu}_s$ as

$$\overline{cu}_s = \frac{\int_{t_0}^{t_0+\Delta T} c_s(t)\,\mathrm{d}t}{\Delta T}. \tag{5}$$

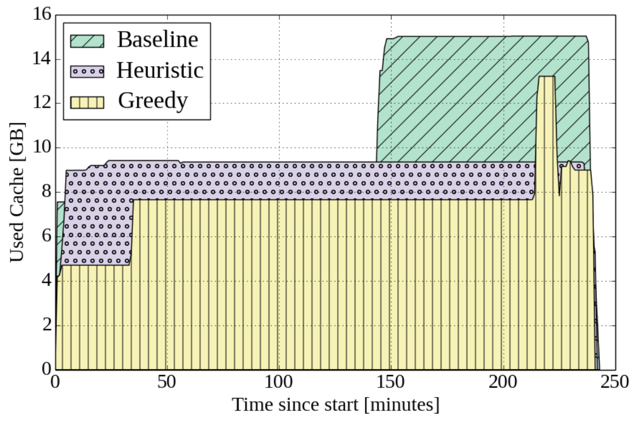
TABLE III. AVERAGE AND MAXIMUM CACHE USAGE FOR DIFFERENT SCHEDULES OF THE TPC-DS-63Q WORKLOAD.

Algorithm	Avg [GB]	Decr.	Max [GB]	Decr.
Baseline	10.439	100%	15.025	100%
Greedy TMB	7.514	71%	13.217	88%
Greedy WTMB	4.430	42%	9.529	63%
Heuristic TMB	9.190	88%	9.586	63%
Heuristic WTMB	7.113	68%	10.511	70%

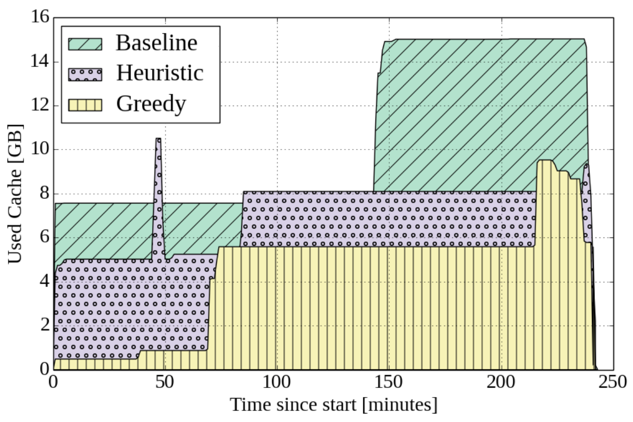
The resulting cache usages are shown in Table III. The schedule of the Greedy algorithm needs on average only 42% of RAM of the Baseline schedule. This leaves either more RAM for other concurrently running applications or results in less read I/O if RAM is scarce.

VI. Conclusion and Future Work

In this paper, we formulated and solved a novel problem in the context of shared data-intensive workload scheduling. Given a set of tasks with data dependencies, such as those corresponding to updating a hierarchy of materialized views, we computed an ordering of the tasks that 1) obeys the precedence constraints encoded by the data dependencies and



(a) Scheduled **without** considering table sizes.



(b) Scheduled considering table sizes.

Fig. 11. Cache usage over time for *tpc-ds-63q*.

2) minimizes cache misses in a cache oblivious setting. Our solution relied on sequencing all tasks that need a particular data item as soon as possible after this item is loaded into the cache. We gave an optimal algorithm based on A* search and efficient heuristics that find nearly-optimal orderings in a fraction of the time needed to run the A* algorithm.

As the experimental results showed, there is still room for improvement of the heuristic algorithm. Another interesting direction for future work is to extend our approach to distributed scheduling of MapReduce jobs.

Acknowledgments

The research leading to these results has received funding from the European Union under the FP7 Grant Agreement n.318627 (Integrated Project mPlane). All members of FTW are supported by the Austrian Government and by the City of Vienna within the competence center program COMET.

References

[1] K. Agrawal, J. T. Fineman: Brief announcement: cache-oblivious scheduling of streaming pipelines. SPAA 2014: 79-81

[2] P. Agrawal, D. Kifer, C. Olston: Scheduling shared scans of large data files. PVLDB 1(1): 958-969 (2008)

[3] M. Ahmad, A. Aboulnaga, S. Babu, K. Munagala: Interaction-aware scheduling of report-generation workloads. VLDB J. 20(4): 589-615 (2011)

[4] A. Allahverdi, C. Ng, T. Cheng, M. Kovalyov: A survey of scheduling problems with setup times or costs. European Journal of Operational Research 187(3):985-1032, 2008

[5] S. Arumugam, A. Dobra, C. Jermaine, N. Pansare, L. Perez: The DataPath system: a data-centric analytic processing engine for large data warehouses. SIGMOD 2010: 519-530

[6] A. Bär, A. Finamore, P. Casas, L. Golab, M. Mellia: Large-Scale Network Traffic Monitoring with DBStream, a System for Rolling Big Data Analysis. IEEE BigData 2014: 165-170

[7] C. Chekuri, R. Motwani: Precedence Constrained Scheduling to Minimize Weighted Completion Time on a Single Machine. Discrete Applied Mathematics 98(1-2):29-38 (1999)

[8] P. Crescenzi, V. Kann: A compendium of NP optimization problems, 1998. ftp://ftp.nada.kth.se/Theory/Viggo-Kann/compendium.pdf

[9] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, L. Sheng: Optimizing Refresh of a Set of Materialized Views. VLDB 2005: 1043-1054

[10] M. Frigo, C. Leiserson, H. Prokop, S. Ramachandran: Cache-Oblivious Algorithms. FOCS 1999: 285-297

[11] G. Giannikis, G. Alonso, D. Kossmann: SharedDB: Killing One Thousand Queries With One Stone. PVLDB 5(6):526-537 (2012)

[12] G. Giannikis, D. Makreshanski. G. Alonso, D. Kossmann: Shared Workload Optimization. PVLDB 7(6):429-440 (2014)

[13] M. Garey, D. Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness, 1979, W. H. Freeman & Co.

[14] M. Garey, D. Johnson, L. Stockmeyer: Some simplified NP-complete graph problems. Theoretical Computer Science 1(3):237-267 (1976)

[15] L. Golab, T. Johnson, J. S. Seidel, V. Shkapenyuk: Stream warehousing with DataDepot. SIGMOD 2009: 847-854

[16] L. Golab, T. Johnson, V. Shkapenyuk: Scalable Scheduling of Updates in Streaming Data Warehouses. TKDE 24(6):1092-1105 (2012)

[17] A. Gupta, S. Sudarshan, S. Viswanathan: Query Scheduling in Multi Query Optimization. IDEAS 2001: 11-19

[18] W. Labio, R. Yerneni, H. Garcia-Molina: Shrinking the Warehouse Update Window. SIGMOD 1999: 383-394

[19] W. Lehner, R. Cochrane, H. Pirahesh, M. Zaharioudakis: fAST Refresh using Mass Query Optimization. ICDE 2001: 391-398

[20] G. Luo, J. F. Naughton, C. J. Ellmann, M. Watzke: Transaction reordering. DKE 69(1): 29-49 (2010)

[21] H. Mistry, P. Roy, S. Sudarshan, K. Ramamritham: Materialized View Selection and Maintenance Using Multi-Query Optimization, SIGMOD 2001: 307-318.

[22] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, N. Koudas: Sharing across Multiple MapReduce Jobs. ACM Trans. Database Syst. 39(2): 12 (2014)

[23] C. Papadimitriou: The NP-completeness of the Bandwidth Minimization Problem, Computing, 16(3):263-270 (1976)

[24] I. Psaroudakis, M. Athanassoulis, A. Ailamaki: Sharing Data and Work Across Concurrent Analytical Queries, PVLDB 6(9):637-648 (2013)

[25] D. Tsaih, G. Wu, C. Chang, S. Hung, C. Wu, and H. Lin: An Efficient A* Algorithm for the Directed Linear Arrangement Problem. WSEAS Transactions on Computers, 7(12):1958-1967 (2008)

[26] J. L. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.-L. Wu, R. Vernica: On the optimization of schedules for MapReduce workloads in the presence of shared scans. VLDB J. 21(5): 589-609 (2012)

[27] M. Zukowski, S. Héman, N. Nes, P. A. Boncz: Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. VLDB 2007: 723-734