Optimizing Index Deployment Order for Evolving OLAP *

Hideaki Kimura[+] Carleton Coffrin[+] Alexander Rasin[†] Stanley B. Zdonik[+]
[+]{hkimura, cjc, sbz}@cs.brown.edu, Brown University, Providence, RI
[†]arasin@cdm.depaul.edu, DePaul University, Chicago, IL

ABSTRACT

Many database applications deploy hundreds or thousands of indexes to speed up query execution. Despite a plethora of prior work on index selection, designing and deploying indexes remains a difficult task for database administrators. First, real-world businesses often require online index deployment, and the traditional *off-line* approach to index selection ignores intermediate workload performance during index deployment. Second, recent work on *on-line* index selection does not address effects of complex interactions that manifest during index deployment.

In this paper, we propose a new approach that incorporates transitional design performance into the overall problem of physical database design. We call our approach *Incremental Database Design*. As the first step in this direction, we study the problem of *ordering* index deployment. The benefits of a good index deployment order are twofold: (1) a prompt query runtime improvement and (2) a reduced total time to deploy the design. Finding an effective deployment order is difficult due to complex index interaction and a factorial number of possible solutions.

We formulate a mathematical model to represent the index ordering problem and demonstrate that Constraint Programming (CP) is a more efficient solution compared to other methods such as mixed integer programming and A* search. In addition to exact search techniques, we also study local search algorithms that make significant improvements over a greedy solution with minimal computational overhead.

Our empirical analysis using the TPC-H dataset shows that our pruning techniques can reduce the size of the search space by many orders of magnitude. Using the TPC-DS dataset, we verify that our local search algorithm is a highly scalable and stable method for quickly finding the best known solutions.

1. INTRODUCTION

The *selection* and *deployment* of indexes has always been one of the most important roles of database administrators (DBAs). Both industry and academia have intensively focused their study on the automatic selection of indexes in physical database design [1, 24]. Every modern commercial database management system (DBMS) ships an automatic design tool as its key component. These design tools support DBAs by suggesting sets of indexes that dramatically improve query execution.

However, recent software mandates complex data processing over hundreds or thousands of tables, making the selection of an appropriate set of indexes impossible for human DBAs and extremely challenging for automated tools. Furthermore, modern enterprises demand operational business intelligence, where always-on data warehouses support complex analytic queries over continuously evolving datasets. Last but not least, the queries, data, and even schema in very large data-warehouses are continually evolving. The main reason for this is two fold.

- Several iterations are often required to accurately translate business requirements into database schema.
- Businesses dynamically change their requirements. As a result, they have to continuously collect and analyze new kinds of data for timely decisions.

This problem has been studied as *schema evolution* [4] mainly for logical table schema designs. For large data-warehouses, frequently running the off-line tools and deploying all the suggested indexes is impractical.

One emerging approach for solving this problem is the *online-index selection* [5, 20]. The main idea is to keep monitoring the queries on the database to deploy (or drop) appropriate indexes when it sees a shift in query workload. The online approach can quickly react to the change in the database. Furthermore, the sequence of small deployments will adaptively lead to an optimized state of the data-warehouse over time.

Although the online approach is a great step towards optimization for dynamically shifting workloads, it has limitations too. By its nature, the online index selection approach selects a single or a small number of indexes at a time. If it is necessary to deploy several indexes on related tables together to speed-up queries, the approach is not likely to select them, yielding in local optima. This problem arises because, not only query workloads, but also the logical table schema is changing. Even a small change in business requirements sometimes requires drastically different queries as well as logical and physical design.

For example, imagine a popular online digital music shop, *i***Z**unes Store. Hundreds of millions of customers are registered in a table ***CUSTOMER (CUSTID, NAME, ADDRESS, COUNTRY, . . .)***. The table is currently clustered by its dimensional attribute ***COUNTRY*** because the company's analysts' roll-up reports are categorized by the customers' countries of residence. The company has received an outpouring of complaints from customers that it is quite inconvenient that they need to create and switch between multiple ac-

*An extended version of this paper is available at http://arxiv.org/abs/1107.3606

EDBT '12 Berlin, Germany

counts to buy music from localized versions of iZunes Store in different countries. Thus, the company decided to tie each customer to multiple countries. To accomodate this small change, the logical database schema evolved to add a new *n:n* table ***CUST_COUNTRIES(CUSTID, COUNTRY)***, eliminating ***COUNTRY*** from ***CUSTOMER***.

This schema change requires the analysts to modify many, perhaps all, of their reports. Moreover, with regard to the physical schema, all clustered and secondary indexes on ***CUSTOMER*** must be drastically re-designed, as well as the materialized views joining the table with related dimensions.

An online index approach cannot capture the impact of such a change. The common approach in online index selection is to precompute a set of potentially beneficial indexes and only re-evaluate their benefits for given representative queries [5, 20]. This method does not work well in the aforementioned situation because an entirely different set of indexes must be considered.

Further, their selection algorithm often does not consider complex interactions between indexes. To process complex multi-join analytic queries, it is often required to deploy more than 10 indexes simultaneously. There are also interactions that speed-up index creation, which require optimizing the *order* of index deployment studied in this paper. Exploiting the complex index interactions requires a detailed analysis over millions of queries and thousands of candidate indexes, which is impossible in on-line design tools.

The root problem is that their selection algorithm must be as low-overhead as possible in order to continuously monitor query workloads and quickly react to a shift of workload.

1.1 Incremental Database Design

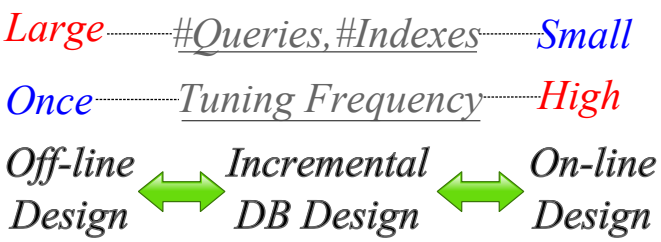

Figure 1: Incremental Database Design

Motivated by the observations above, we have started exploring a new approach positioned between the two extremes; off-line and on-line as illustrated in Figure 1. Our target is a very large data-warehouse which needs a drastic change in its physical design. It will frequently need design changes, so it is necessary to consider not only the query runtime but the deployment time of suggested indexes to incrementally evolve along with the business requirement. On the other hand, the change is relatively less frequent (e.g., a week) than what on-line index approach is targeting (e.g., minutes or hours). This allows us to employ more sophisticated analysis on the choice of indexes and their deployment schedule.

We call this new type of database design tool as ***Incremental Database Design*** (IDD) and are studying its requirements, design and implementation as a long term project.

One interesting use case of IDD is the real-time recovery. Nowadays, it is becoming common to deploy a large data-warehouse over a number of commodity machines. In such a system, a node failure necessitates recovering the part of indexes and materialized views stored in the node. In this case, the DBA can use an IDD tool to complement the performance degradation because of the lack of indexes as soon as possible.

The first challenge towards this direction, which this paper mainly

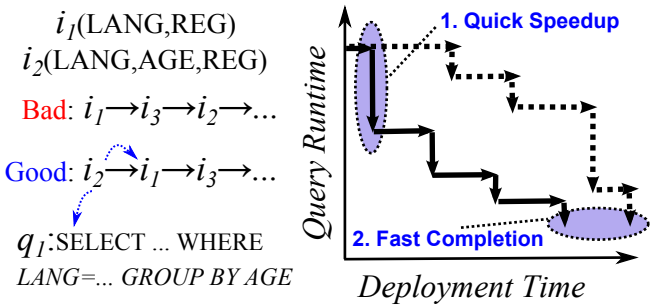

Figure 2: Index Deployment Orders: Good vs. Bad

studies, is how to schedule the deployment of indexes to quickly complete the deployment or achieve the majority of query speed-ups. As mentioned at the beginning of this section, the *deployment* of indexes is an important aspect of database maintenance. Deploying indexes is a very costly operation and DBAs give it as much care and attention as possible. It consumes immense hardware resources and takes a long time to complete on large tables.

Moreover, it is likely that a database requires hundreds of indexes to be deployed due to the growing number and complexity of queries and schema. For example, a database designer built in a commercial DBMS suggested 148 indexes for the TPC-DS benchmark which took more than 24 hours to be deployed in the DBMS even with the smallest (Scale-100) instance.

1.2 Index Deployment Order

We observed that during the long process of deploying many indexes over large databases, the order (sequence) of index deployment has two significant impacts on user benefits, illustrated in Figure 2. First, a good order achieves prompt query runtime improvements by deploying indexes that yield greater query speed-ups in early steps. For example, an index that is useful for many queries should be created first.

Second, a good order reduces the deployment time by allowing indexes to utilize previously built indexes to speed up their deployment. For instance, the index i_1 (LANG, REGION) should be made *after* the wider index i_2 (LANG, AGE, REGION) to allow building from the index, not the table. We observe in the TPC-DS case that a good deployment order can reduce the build cost of an index up to 80% and the entire deployment time as much as 20%.

Despite the potential benefits, obtaining the optimal index order is challenging. Unlike typical job sequencing problems [3], both the benefit and the build cost of an index are dependent on the previously built indexes because of **index interactions**. These database specific properties make the problem non-linear and much harder to solve. Also, as there are $n!$ orderings of n indexes, a trivial exhaustive search is intractable, even for small problems.

One prevalent approach for optimization problems is to quickly choose a solution by a greedy heuristic. However, the quality of a greedy approach can vary from problem to problem and has no quality guarantee. Another popular approach is to employ exact search algorithms such as A* or mixed integer programming (MIP) using the branch-bound (BB) method to prune the search space. However, the non-linear properties of the index interactions yield poor linear relaxations for the BB method and both MIP and A* degenerate to an exhaustive search without pruning.

2. OVERVIEW

In this paper, we formally define the ordering problem as a mathematical model and propose several pruning techniques not based

on linear relaxation but on the combinatorial properties of the problem. We show that these problem specific combinatorial properties can reduce the size of the search space by many orders of magnitude. We solve the problem using several techniques including, Constraint Programming (CP) and MIP, and show that this kind of problem is easiest to model and has better performance in a CP framework. We then extend the CP model using local search methods to get high quality solutions very quickly for larger problems. We evaluate several local search methods and devise a variable neighborhood search (VNS) method building on our CP model that is highly scalable and stable.

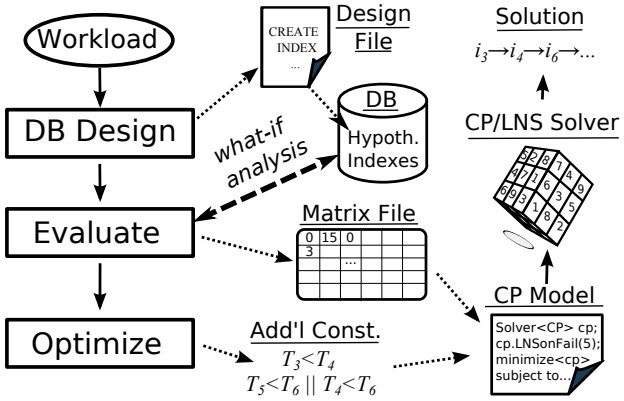

Figure 3: Solution Overview

Figure 3 gives an overview of our CP-based solution for the index ordering problem. Given a query workload, we first run a physical database design tool to obtain a set of suggested indexes. Then, we analyze the indexes. To avoid actually creating indexes, we use *what-if* [7] interface of the DBMS to hypothetically create each index and evaluate its benefits using the query optimizer. The result is a matrix which stores the benefits and creation costs of all indexes as well as the interactions between them. When formulating this matrix as CP code, we also derive additional constraints to speed up the CP solver. The CP/LNS engine then solves the problem and produces the optimized index deployment order.

To explain each piece of the solution, this paper is organized as follows. Section 3 reviews the related work. Section 4 formally defines the problem of index deployment. Section 5 provides several techniques to efficiently solve the problem. Section 6 describes our CP model for the problem. Section 7 extends the CP model with local search to solve larger problems. Section 8 reports the experimental results and Section 9 concludes this paper and discusses our next step towards incremental database design based on this work.

In summary, our contributions are:

- Vision of incremental database design
- A formal description of the index deployment order problem
- Problem specific properties to reduce problem difficulty
- Models and algorithms for Greedy, MIP, CP and Local Search
- Analysis of various solution techniques and solvers
- Empirical analysis on TPC-H and TPC-DS.

To the best of our knowledge, this work is the first to study CP methods in the context of physical database design despite its significant potential as an accurate and scalable design method.

3. RELATED WORK

3.1 Physical Database Design

Because of the complexity of query workloads and database mechanics, no human database administrator (DBA) can efficiently select a set of database objects (e.g., indexes) subject to resource constraints (e.g., storage size) to improve query performance. Hence, significant research effort has been made both in academia and in industry to automate the task of physical database design [8, 24].

The AutoAdmin project [1] pioneered this field by implementing the *what-if* method [7, 10] which creates a set of potentially beneficial indexes as hypothetical indexes to evaluate their expected benefit by the database's query optimizer.

Once the benefits of each index are evaluated, the problem of database design is essentially a boolean knapsack problem, which is NP-hard. The database community has tried various approaches to solve this problem. The most common approach is to use greedy heuristics based on the benefit of indexes [8] or on their density [24] (benefit divided by size). However, a greedy algorithm is not assured to be optimal and could be arbitrarily bad in some cases. Hence, some research has explored the use of exact methods such as mixed integer programming (MIP) [17, 19] and A* search [13].

Despite the wealth of research in physical database design, the problem of optimizing index deployment order has not been studied closely. Practically all prior work in this field considers both the query workloads and the indexes as a *set*. One exception is [2] which considers a query workload as a sequence, but only considers dropping and re-creating existing indexes to reduce maintenance overhead. The work in [21] had also considered ordered deployment, but primarily as a way to greedily speed up queries at every step, rather than optimize the overall index deployment sequence. Bruno et al. [6] mentioned a type of ordering problem as an unsolved problem, but their objective does not consider prompt query speed-ups. Also, they only suggested to use A* or Dynamic Programming and did not solve the problem in [6].

3.2 Online Index Selection

Schnaitter et al. proposed the COLT framework [20] which progressively deploys (or drops) indexes as the current dominant query workload changes. Their approach controls the online tuner overhead through clustering similar queries in the workload and a (user-specified) bound on the number of optimizer calls per tuning iteration. However, their designs are limited to single-column indexes due to the high complexity of the problem. Moreover, to further simplify the problem they assume that the benefit of each candidate index is completely independent. In practice, this is rarely a realistic assumption, particularly when candidate multi-column indexes are considered.

In [5], Bruno et al. propose a similar mechanism that tracks newly arriving queries, gathering the potential benefit of hypothetical candidate indexes. Once it appears that the cost of adding a new index is justified by the anticipated query runtime improvement, the new index is introduced into the physical configuration. The algorithm proposed in [5] can add several indexes at once, but it does not choose a particular deployment order. It is also aware of possible index interactions, but uses a rudimentary syntactic estimate based on column overlap between indexes (again, due to high problem complexity and potential algorithm overhead).

The work in [21] presented a framework for detecting and evaluating the relative degree of index interaction as it affects query performance. The authors have suggested using a visualization mechanism to assist the DBA decisions by identifying which of the candidate indexes have strong interactions. They have furthermore

proposed an index deployment utility function that is very similar to the one we describe in Section 4.1. However, their solution to index deployment problem is ultimately a greedy selection of indexes from the set chosen by the DBA. Although they propose using dynamic programming to achieve a better deployment ordering, that approach has a number of shortcomings, such as failing to account for the cost to build each index and the way index interaction affects this cost.

Although some have explored the problem of index benefit interaction [5, 21] in their work, they chose to approximate the index interaction benefit to avoid invoking the query optimizer too frequently. They made this choice in order to contain the cost of the online algorithms and in order to quickly respond to shifts in user workload. While such approach allows for agile database tuning, it tends to deploy very few indexes at a time. Thus it effectively ignores the problem of order of index deployment, instead always choosing the best one (or best few) indexes at each deployment iteration. Furthermore, to our knowledge no one has yet considered incorporating the effects of interactions between indexes as they affect the cost of index building itself. As we explain in Section 4.2, such interaction can have a significant effect on the overall index deployment cost. In this work, we use the exact query optimizer cost estimates to evaluate index interaction and consider the potential effects on the cost of building the indexes as they are deployed. We use CP (as a superior alternative to a greedy or MIP approach) and incorporate a number of carefully defined index interaction rules (see Section 4.2) to find a good index deployment order.

In this work, we assume that creation of a single index is an atomic process as that is the default DBMSs behavior. A alternative approach explored in [14, 18] is to build the index piece-by-piece. The work in [18] explores the idea of building the index concurrently with table updates. They also propose the idea of querying the incomplete index, provided the query can be answered using the part of the index that was already built. A similar idea [14], explored in the context of a column store DBMS, is to copy and reorganize the data content as the queries access it. This approach provides more immediate adaptation to the changes in query workload and can also recycle the work already performed by the query. In this paper, however, we do not assume these advanced functionalities built in the DBMS.

3.3 Branch-and-Bound

All decision problems, such as the index order problem, can be formulated as tree search problems. Such a tree has one level for each decision that must be made and every path from the root node to a leaf node represents one solution to the problem. In this way, the tree compactly represents all the possible problem solutions. However, exploring this entire tree is no more tractable than exhaustive search. Therefore, many tree search techniques have been developed to more efficiently explore the decision tree.

Branch-and-Bound (BB) is a tree search method which prunes (a.k.a. removes) sub-trees by comparing a lower bound (best possible solution quality) with the current best solution. A* is a popular type of BB search method which uses a user-defined heuristic distance function to deduce lower bounds.

MIP solvers, such as IBM ILOG CPlex, are also based on BB. MIP uses a linear relaxation of the problem to deduce lower bounds, and the pruning power of the MIP is highly dependent on the tightness of the linear relaxation.

BB is efficient when the relaxation is strong, however it degrades as the relaxation becomes weaker, which is often the case for non-linear problems. Also, MIP only supports linear constraints, and it is tedious to model non-linear properties using only linear constraints.

Table 1: MIP and CP Comparison

	MIP	CP
Constraints & Objectives	Linear Only	Linear & Non-Linear
Pruning Method	Branch-Bound & Linear Relaxation	Branch-Prune & Custom Constraints
Non-Exhaustive Search Variant	N/A (Best Solution)	Local Search
Best Suited for	Linear Problems	Combinatorial Problems

3.4 Constraint Programming

Similar to MIP, Constraint Programming (CP) does a tree search over the values of the decision variables. Given a model, a CP solver explores the search tree like a MIP solver would. However, there are a few key differences summarized in Table 1.

First, CP uses a branch and prune (BP) approach instead of BB. At each node of the tree, the CP engine uses the combinatorial properties of the model's constraints to deduce which branches cannot yield a higher quality solution. Because the constraints apply over the combinatorial properties of the problem, the CP engine is well suited for problems with integer decision variables. Instead of a linear relaxation to guide the search procedure in MIP, CP models often include specialized search strategies that are designed on a problem-by-problem basis [3].

Second, CP does not suffer from the restriction of linearity that MIP models have. This is especially helpful for our problem which has a non-linear objective function and constraints such as nested decision variable indexing.

Third, CP models allow a seamless extension to local search. When the problem size becomes so large that proving a solution's optimality is impossible, the goal becomes getting a near-optimal solution as fast as possible. In this setting, global search techniques (such as MIP and CP) often become impractical because they exhaustively search over every sub-tree that has some chance of containing the optimal solution regardless of how slight the chance is, and how large the sub-tree is. Such exact methods are thus inappropriate to quickly find high quality solutions. On the other hand, local search on top of CP such as Large Neighborhood Search (LNS) [23] combines the pruning power of CP with the scalability of local search.

In later sections, we will contrast these differences more vividly with concrete case studies for modeling and solving the index order problem. Although we find that CP is highly effective for physical database design, to the best of our knowledge this is the first time that CP has been applied to this problem domain.

4. PROBLEM DEFINITION

This section formally defines the index deployment order problem. Throughout this section, we use the symbols, constant values, and decision variables listed in Table 2 and 3. Please note that although we refer to indexes throughout this paper, any auxiliary database structure that speeds up query performance (e.g., MV) can be trivially incorporated into our formulation.

4.1 Objective Values

Every feasible solution to the problem is a *permutation* of the indexes. An example permutation of indexes $\{i_1, i_2, i_3\}$ is $i_3 \rightarrow i_1 \rightarrow i_2$. As discussed in the introduction, we want to achieve a prompt query runtime improvement and a reduction in total

deployment time. Hence, the metric we define to compare solutions is the area under the improvement curve illustrated in Figure 4. This area is defined by $\sum_i (R_{i-1}C_i)$, the summed products of the **previous** total query runtime and the cost to create the i^{th} index. The previous total query runtime is used because the query speed-up occurs only after we complete the deployment of an index.

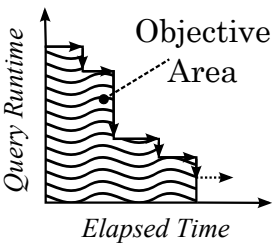

Figure 4: Objective Values

Table 2: Symbols & Constant Values (in lower letters)

$i \in I$	An index. $I = \{i_1, i_2, .., i_{\|I\|}\}$
$q \in Q$	A query.
$p \in P$	A query plan (a set of indexes).
$plans(q) \in P$	Feasible query plans for query q.
$qtime(q)$	Original runtime of query q.
$qspdup(p, q)$	Speed-up of using plan p for query q compared to the original runtime of q.
$ctime(i)$	Original creation cost of index i.
$cspdup(i, j)$	Speed-up of using index j for index i.

Because we would like to reduce the query runtimes and total deployment time, the smaller the area the better the solution. Thus, this objective function considers prompt query speed-ups and total deployment time simultaneously.

4.2 Index Interactions

This section describes the various **index interactions**, which make the problem unique and challenging.

Competing Interactions: Unlike typical job sequencing problems, completing a job (i.e. building an index) in this problem has varying benefits depending on the completion time of the job.

This is because a DBMS can only use one query execution plan at a time. Consider the indexes $i_1(City)$ and $i_2(City, Salary)$ from the following query:

```
SELECT AVG(Salary) FROM People
WHERE City=Prov
```

Assume the query plan using i_1 is 5 seconds faster than a full scan while the plan using the covering index i_2 is 20 seconds faster.

The sequence $i_1 \rightarrow i_2$ would have a 5 second speed-up when i_1 is built, and only $20 - 5 = 15$ second speed-up when i_2 is built because the query optimizer in the DBMS picks the fastest query plan possible at a given time, removing the benefits of suboptimal query plans. Likewise, the sequence $i_2 \rightarrow i_1$ would observe no speed-up when i_1 is built. We call this property *competing interactions* and generalize them by constraint 3 in the mathematical model.

Query Interactions: It is well known that two or more indexes together can speed up query execution much more than each index alone. Suppose we have two indexes $i_1(City)$ and $i_2(EmpID)$ for the following query:

```
SELECT .. FROM People p1 JOIN People p2
ON (p1.ReportTo=p2.EmpID) WHERE p1.City=Prov
```

A query plan using one index ($\{i_1\}$ and $\{i_2\}$) requires a table scan for the JOIN and costs as much as the no-index plan $\{\emptyset\}$. A query plan using both i_1 and i_2 ($\{i_1, i_2\}$) avoids the full table scan and performs significantly faster. We call such index interactions *query interactions*. Because of such interactions, we need to consider the speed-ups of the three query plans separately, rather than simply summing up the benefits of singleton query plans.

Build Interactions: As a less well known interaction, some indexes can be built faster if there exists another index that has some overlap with the keys or included columns of the index to be built.

Table 3: Decision Variables (in capital letters)

$T_i \in \{1, ..\|I\|\}$	The position of index i in the deployment order. T is a permutation of $\{1, 2, .., \|I\|\}$.
R_i	Total query runtime after i^{th} index is built.
$X_{q,i}$	q's speed-up after i^{th} index is built.
$Y_{p,i} \in \{0, 1\}$	Whether p is available after i^{th} index is built.
C_i	Cost to create i^{th} index.

For example, $i_1(City)$ and $i_2(City, Salary)$ have interactions in both ways. If i_2 already exists, building i_1 becomes substantially faster because it requires only an index scan on i_1 rather than scanning the entire table. On the other hand, if there already is i_1, building i_2 is also faster because the DBMS does not have to sort the entire table. We call these index interactions *build interactions* and generalize it by constraint 5 in the mathematical model.

This means that the index build cost is not a constant in our problem but a variable whose value depends on the set of indexes already built. Bruno et al. [6] also mentioned this effect earlier. In Section 8 we show there exist a rich set of such interactions.

Precedence: Sometimes, an index *must* precede some other indexes. One example is an index on a *materialized view* (MV). A MV in a certain type of DBMS is created when its clustered index is built. Non-clustered (secondary) indexes on the MV cannot be built before the clustered index. Hence, the clustered index must precede the secondary indexes on the same MV in a feasible solution for such a DBMS.

Another example is a secondary index that exploits *correlation* [16]. For example, SQL Server supports the *datetime correlation optimization* which exploits correlations between clustered and secondary datetime attributes. To work properly, such an index requires the corresponding clustered index to be built first.

Detection: Some prior work explored a way to efficiently find such interacting indexes [21]. In our experiments, we detect interactions by calling the query optimizer with hypothetical indexes as detailed in Section 8.

4.3 Mathematical Model

Embodying the concepts of index interactions discussed above, the full mathematical model is defined as follows,

$$\text{Objective:} \qquad min \sum_i (R_{i-1}C_i) \tag{1}$$

$$\text{Subject to:} \qquad Y_{p,i} = \{T_j \leq i : \forall j \in p\} : \forall p, i \tag{2}$$

$$X_{q,i} = \max_{p \in plans(q)} qspdup(p, q)Y_{p,i} : \forall q, i \tag{3}$$

$$R_i = \sum_q (qtime_q - X_{q,i}) : \forall i \tag{4}$$

$$C_{T_i} = ctime(i) - \max_{j:T_j<T_i} cspdup(i, j) : \forall i \tag{5}$$

(2) states that a query plan is available only when all of the indexes in the query plan are available. (3) calculates the query speed-up by using the fastest query plan for the query at a given time. (4) sums up the speed-ups of each query and subtract from the original query runtime to get the current total runtime. (5) calculates the cost to create index i (C_{T_i} because C is indexed by the order) by considering the fastest available ($T_j < T_i$) interaction. For simplicity, this constraint assumes every build interaction is pair-wise (one index helps one other index). So far we have ob-

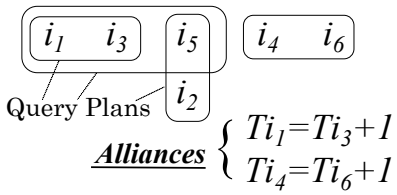

Figure 5: Alliances

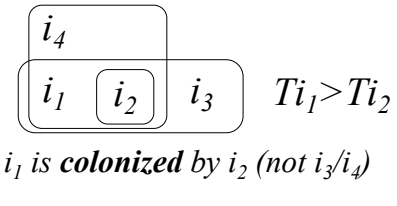

Figure 6: Colonized Indexes

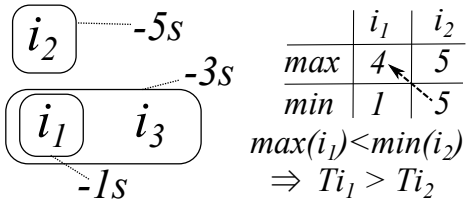

Figure 7: Dominated Indexes

served this to be the case, but this constraint can easily be extended for arbitrary interactions by doing a similar formulation using X and Y variables.

Given this mathematical formulation, our goal is to find the permutation with the minimal objective value and prove its optimality. However, for large problems where an optimality proof is intractable, we are satisfied with any solution that can be found quickly and makes a significant improvement over a greedy solution technique.

4.4 Discussion

In formalizing a problem as rich as the index deployment order problem there are many choices to be made. One option is to simplify the problem to have some nice theoretical properties, such as good approximation algorithms and tight lower bounds. Another approach is to include as much sophistication as possible in the problem formulation so that it might be deployable in industrial applications. By choosing to include all of the index interactions and a complex objective function, this work has chosen the later option. In doing so, tight lower bounds and theoretical guarantees are outside the scope of this formulation. Hopefully this short coming is balanced by broader industrial applications. In fact, the experimental results demonstrate that index interactions are an important consideration to this problem and removing them would have a significant effect on solution quality. Recognizing that theoretical guarantees are out of reach, this work will conduct a rigorous experimental study to understand the performance of several solution techniques for the index deployment order problem, and focus on the scale of problems that are necessary for industrial deployment.

The objective function is another area of many choices. For example, putting different weights on particular queries can be incorporated by simply scaling up or down runtimes of the queries. Or, one can consider minimizing the total deployment time, $\sum C_i$, like [6]. In either case, most of the modeling and pruning strategies in this paper will be usable with minor modifications.

5. PROBLEM PROPERTIES

This problem has up to $|I|!$ possible solutions. An exhaustive search method that tests all the solutions is intractable even for small problems. In this section, we analyze the combinatorial properties of the problem. Based on the problem specific structure, such as index interactions, we established a rich set of pruning techniques which significantly reduce the search space. This section describes the intuition behind each optimization technique and how we apply it to the problem formulation. The formal proofs and cost analysis of each technique can be found in the extended version of the paper [15].

These techniques are inherent properties of the problem which are independent of a particular solution procedure. In fact, we demonstrate that these techniques reduce the runtime of both MIP and CP solvers by several orders of magnitude in Section 8.

5.1 Alliances

The first problem property is an *alliance* of indexes that are always used together. We can assume that such a set of indexes are always created together.

Figure 5 exemplifies alliances of indexes. The figure illustrates 4 query plans with 6 indexes; $\{i_1, i_3\}$, $\{i_1, i_3, i_5\}$, $\{i_2, i_5\}$, $\{i_4, i_6\}$. Observe that i_1 and i_3 always appear together in all query plans they participate in. Therefore, creating only one of them gives no speed-up for any query. This means we should **always** create the two indexes together. Hence, we add a constraint $T_{i_1} = T_{i_3} + 1$. Same to i_4 and i_6. Note that i_2 and i_5 are **not** an alliance because i_5 appears in the query plan $\{i_1, i_3, i_5\}$ without i_2. An alliance is often a set of strongly interacting indexes each of which is not beneficial by itself. An alliance of size n essentially removes $n-1$ indexes and substantially simplifies the problem.

5.2 Colonized Indexes

The next problem property is a *colonized* index which is a one-directional version of alliances. If all interactions of an index, i, contain another index, j but not vice versa, then i is called a colonized index and should be created after j.

Figure 6 shows a case where i_1 is colonized by i_2. i_1 always appears with i_2 in all query plans i_1 participates, but not vice versa because there is a query plan that only contains i_2.

In such a case, creating i_1 alone always yields no speed-up. On the other hand, creating i_2 alone might provide a speed-up. Thus, it is always better to build the colonizer first; $T_{i_1} > T_{i_2}$.

Observe that i_1 is not colonized by i_3 or i_4 because i_1 appears in plans where only one of them appears. In fact, if the plan $\{i_1, i_2, i_4\}$ is highly beneficial, the optimal solution is $i_2 \rightarrow i_4 \rightarrow i_1 \rightarrow i_3$, so $T_{i_1} > T_{i_3}$ does not hold. Likewise, if the plan $\{i_1, i_2, i_3\}$ is highly beneficial, the optimal solution is $i_2 \rightarrow i_3 \rightarrow i_1 \rightarrow i_4$, so $T_{i_1} > T_{i_4}$ does not hold.

5.3 Dominated Indexes

The next problem property is called a *dominated* index which is an index whose benefits are **always** lower than benefits of another index. Dominated indexes should always be created last.

To simplify, consider the case where indexes have the same build cost and every query plan is used for different queries. For the full formulation without these simplifications, see the extended version [15].

Figure 7 depicts an example where i_1 is dominated by i_2. The maximum benefit of an index is the largest speed-up we get by building the index. For example, the maximum benefit of i_1 occurs when there already exists i_3, which is $1 + 3 = 4$ seconds. Conversely, the minimum benefit is the smallest speed-up we get by building the index. i_1's minimum benefit happens when there is no i_3 index; only 1 second. On the other hand, both the maximum and minimum benefits of i_2 are 5 seconds.

Hence, the speed-up of building i_1 is always lower than the speed-up of building i_2. As our objective favors a larger speed-up at an earlier step, we should always build i_2 before i_1; $T_{i_1} > T_{i_2}$.

5.4 Disjoint Indexes and Clusters

The next problem property is called a *disjoint* index, which is

an index that has no interaction with other indexes. Such indexes do not give or receive any interaction to affect the build time and speed-up and sometimes we can deduce powerful constraints from them. Figure 8 shows an example of a disjoint index i_4 and a *disjoint cluster* $M_1 = \{i_1, i_2, i_3\}$ which has no interaction with other indexes except the members of the cluster.

Suppose we already have a few additional constraints that define the relative order of $\{i_1, i_2, i_3\}$ is $i_1 \rightarrow i_2 \rightarrow i_3$ and we need to insert i_4 into the order. Among the four possible locations for i_4, we can uniquely determine the best place, which we call the *dip*.

We know the placement of i_4 does not affect the build cost and the speed-up of any index in M_1 because i_4 and M_1 are disjoint. In such a case, we should place i_4 after an index whose *density* (the gradient of the diagonal line; speed-up divided by build cost) is larger than i_4's density and before an index with a smaller density. Otherwise, we can improve the order by swapping i_4 with another index because the shaded area in Figure 8 becomes larger when we build an index with a smaller density first. In the example, the best place is between i_2 and i_3, which means $den_{i_1+i_2} > den_{i_4}$, $den_{i_2} > den_{i_4}$ and $den_{i_4} > den_{i_3}$ where den_x is the density of x. We call this location, the dip and there is always exactly one dip.

We can generalize the above technique for non-disjoint indexes when they have special properties which we call *backward-disjoint* and *forward-disjoint*. Consider two disjoint clusters M_i and M_j which contain index i and j respectively. In order to determine whether i precedes or succeeds j in the complete order, we can investigate the interacting indexes of i and j.

i is said to be backward-disjoint regarding j when all interacting indexes of i and j are built after i **or** before j. Conversely, i is said to be forward-disjoint regarding j when all interacting indexes are built before i **or** after j, in other words when j is backward-disjoint regarding i. A disjoint index is both backward and forward disjoint regarding every other disjoint index. Initially most indexes have no disjoint properties, but with the additional constraints from other properties they often become backward or forward disjoint.

An intuitive description of i being backward-disjoint regarding j is that i and j behave as disjoint indexes when we are considering a subsequence $j \rightarrow X \rightarrow i$ for arbitrary X, so i is *disjoint in a backwards order*. Because of the property of disjoint indexes, the subsequence must satisfy $den_i < den_j$ if it is an optimal solution. Thus, if we know $den_i > den_j$, we can prune out all solutions that build j before i. Conversely, if i is forward-disjoint and $den_i < den_j$, then i always succeeds j.

5.5 Tail Indexes

Because of the inequality constraints given by the above properties, sometimes a single index is uniquely determined to be the last (*tail*) index. In that case, we can eliminate the index from the problem for two reasons. First, the last index cannot cause any interaction to speed up other indexes either in query time or build time because all of them precede the last index. Second, the interactions the last index receives from other preceding indexes do not depend on the order of other indexes; all the other indexes are already built. Therefore, we can remove the last index and all of its interactions from consideration, substantially simplifying the problem.

We can extend this idea even if there are multiple candidates for the last index by analyzing the possible tail index patterns.

For example, in the TPC-H problem solved in Section 8.1, i_1 and i_2 turn out to have many preceding indexes and thus the possible orders of them are n (last), $n-1$ (second to last) and $n-2$ (third to last). All possible patterns of the last 3 tail indexes are listed in Figure 9. It also shows the last part of the objective area (*tail objective*) for the 3 tail indexes in each pattern (the shaded areas). We can

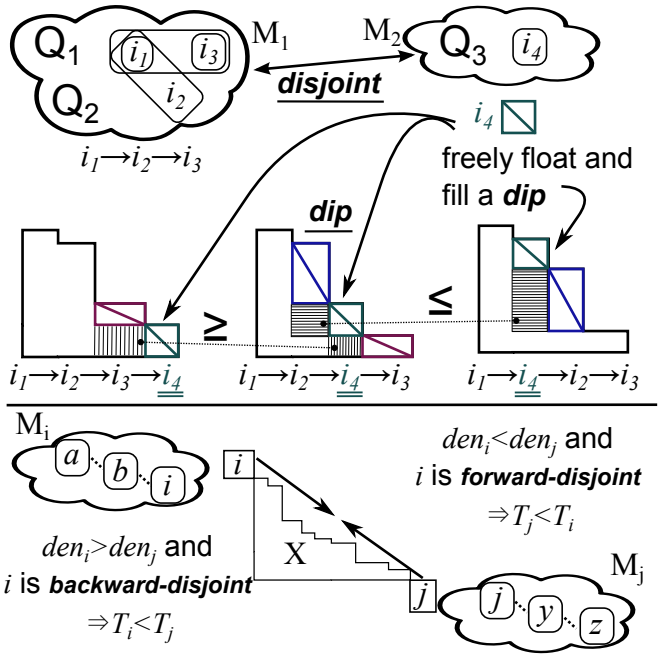

Figure 8: Disjoint Indexes and Disjoint Clusters

calculate the tail objectives because the *set* of preceding indexes is known therefore, regardless of their orders, their interactions to the tail indexes are determined.

Remember that there are many other preceding indexes before the tail indexes. Therefore, we cannot simply compare the tail objectives. For example, the tail objective of $i_2 \rightarrow i_5 \rightarrow i_1$ in Figure 9 is smaller than that of $i_4 \rightarrow i_1 \rightarrow i_2$. However, because the set of preceding indexes is different, we cannot tell if the former tail pattern is better than the latter.

Nevertheless, we can compare the tail objectives if the set of tail indexes is equivalent. $i_4 \rightarrow i_1 \rightarrow i_2$ and $i_1 \rightarrow i_4 \rightarrow i_2$ contain the same set of indexes, thus *the set of preceding indexes is the same too*, which means the objective areas and the order of preceding indexes is exactly the same after we optimize the order of preceding indexes (again, the tail indexes do not affect preceding indexes). Hence, we can determine which tail pattern is better by comparing tail objectives.

Notice that the tail patterns in Figure 9 are grouped by the set of tail indexes and also sorted by the tail objectives in each group. The ones with the smallest tail objective in each group are called the *champion* of the group and they should be picked if the set of indexes are the tails.

Now, observe that i_2 appears as the last index in every champion (in bold font) of all groups. This means i_2 is always the last created index in the optimal deployment order because its tail is always one of the tail champions.

5.6 Iterate and Recurse

We can repeat the tail analysis by fixing i_2 as the last index and considering a sub-problem without i_2. Not surprisingly, we could then uniquely identify i_1 as the second-to-last index.

Furthermore, by removing the determined indexes (and their query plans) and considering the already introduced inequalities, each analysis described in this section can apply more constraints. Therefore, we repeat this process until we reach the fixed-point. This pre-

Tail	Obj.
$\boldsymbol{i_4 \rightarrow i_1 \rightarrow i_2}$	**9.7**
$i_4 \rightarrow i_2 \rightarrow i_1$	9.9
$i_1 \rightarrow i_4 \rightarrow i_2$	12
$\boldsymbol{i_5 \rightarrow i_1 \rightarrow i_2}$	**4.0**
$i_5 \rightarrow i_2 \rightarrow i_1$	4.2
$i_2 \rightarrow i_5 \rightarrow i_1$	4.5
$\boldsymbol{i_8 \rightarrow i_1 \rightarrow i_2}$	**6.8**
$i_8 \rightarrow i_2 \rightarrow i_1$	6.9
$\boldsymbol{i_{11} \rightarrow i_1 \rightarrow i_2}$	**7.1**
$i_{11} \rightarrow i_2 \rightarrow i_1$	7.3

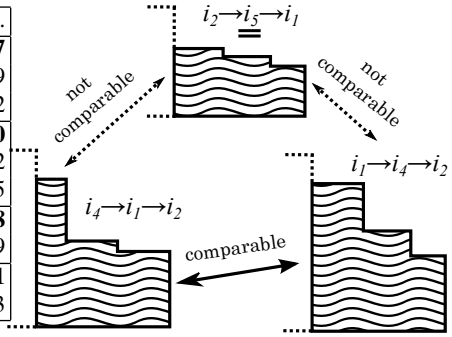

Figure 9: Comparing Tail Indexes of Same Index Set in TPC-H

analysis reduces the size of search space dramatically. In the experimental section, we demonstrate that the additional constraints speed up both CP and MIP by several orders of magnitude.

6. CONSTRAINT PROGRAMMING

In this section, we describe how we translate the mathematical model given in Section 4.3 into a Constraint Programming (CP) model. We then explain how the problem is solved with a CP solver. To illustrate why CP is well suited for this problem, we will compare the CP model to that of MIP throughout this section.

6.1 CP Model

CP allows a flexible model containing both linear and non-linear objectives and constraints. The mathematical formulation presented in Section 4.3 can be modeled in standard CP solvers (e.g., COMET) almost identically, unlike MIP where the model is more obfuscated (an equivalent MIP model is given in the extended version [15]).

$$\text{Objective:} \quad min \sum_i (R[i-1]C[i]) \quad (6)$$

$$\text{Subject to:} \quad alldiffrent(T) \quad (7)$$

$$Y[p,i] = \bigwedge_{j \in p} (T[j] \leq i) : \forall p, i \quad (8)$$

$$X[q,i] = \max_{p \in plans(q)} (qspdup(p,q) Y[p,i]) : \forall q, i \quad (9)$$

$$R[i] = \sum_q (qtime(q) - X[q,i]) : \forall i \quad (10)$$

$$C[T[i]] = ctime(i) - \max_j ((T[j] < T[i]) cspdup(i,j)) : \forall i \quad (11)$$

Objective: Just like the mathematical model, our CP model minimizes the sum of $R[i-1]C[i]$. Although this sounds trivial, MIP cannot directly accept a product of variables (R and C) as an objective.

The most common technique for linearizing a product of variables in MIP is to *discretize* the entire span to a fixed number of uniform timesteps and define the value of each variable at each timestep as an independent variable [22].

However, in addition to losing accuracy, discretization causes severe problems in performance and scalability of MIP which are verified in the experimental section.

***alldifferent* constraint:** The variable T is given in (7) which uses *alldifferent*. This interesting constraint in CP assures all the variables in T are a permutation of their values. The same constraint in MIP would require $|I|^2$ inequalities on elements of T. The CP engine represents it with a *single* constraint which is computationally efficient. This is one of the most vivid examples showing that CP is especially suited for combinatorial problems and how beneficial it is for modeling and optimization purposes.

Logical AND: The AND constraints on Y (2) are translated directly into (8). Although this sounds trivial, again, it is challenging in MIP. Logical AND is essentially a product of boolean variables, which is non-linear, just as the objective was. Modeling such non-linear constraints causes MIP additional overhead and memory consumption as well as model obfuscation.

MIN/MAX sub-problem: The constraints on X (3) which employ the fastest available speed-up for each query are translated directly into (9). Yet again, this is not easy nor efficient in MIP because MIN/MAX is non-linear.

In MIP, this has to be represented as summation of Y and $qspdup$ where only one of Y for each query takes the value of 1 at a given time. Some MIP solvers provide min/max constraint and internally do this translation on behalf of users, but the more severe problem is its effect on performance. When MIP considers the linear relaxation of X, min/max constraint yields little insight. Hence, its BB degenerates to an exhaustive search.

Nested variable indexing: The constraints on C (5) are translated directly into (11). However, this causes two problems in MIP. One is the MIN/MAX as described above, another is the nested variable indexing C_{T_i}. Notice that T is also a variable. Such a constraint cannot be represented in a linear equation. Hence, MIP has to change the semantics of the variable C itself and re-formulate the all of the constraints and the objective calculation.

Additional constraints: Finally, we add the additional constraints developed in Section 5 to reduce the search space.

6.2 Searching Strategy

CP employs branch-prune (BP) instead of BB used by MIP. These two approaches have very different characteristics. In summary, CP is a *white-box* approach with a smaller footprint as opposed to the *black-box* approach of MIP.

Pruning: CP is able to prune the search space by reasoning over the combinatorial properties of the constraints presented in section 6.1. It also utilizes the problem specific constraints we developed in Section 5 to efficiently explore only high quality index orders. Our experimental results demonstrate that combinatorial based pruning is much more effective for this problem than a BB pruning based on a linear relaxation.

Branching: Users *can* and *must* specify how CP should explore the search space. In our case, we found that it is most effective for the search to branch on the $T[i]$ variables and that a *First-Fail* (FF) search procedure was very effective for solving this problem and proving optimality with very small memory footprint.

A FF search is a depth-first search using a dynamic variable ordering, which means the variable ordering changes in each node of the search tree. At each node the variables are assigned by increasing the domain size. Due to the additional constraints, the domains of the $T[i]$ variables vary significantly. This helps the FF heuristic to obtain optimality.

On the other hand, MIP automatically chooses the branching strategy. This is efficient when the linear relaxation is strong, but, when it is not, the BB search degenerates to an exhaustive breadth-first search which causes large memory consumption and computational overhead. In fact, we observe that MIP finds no feasible solution for large problems within several hours and quickly runs out of memory.

7. LOCAL SEARCH

Although CP is well suited for this ordering problem, when there is a large number of indexes with dense interactions between them,

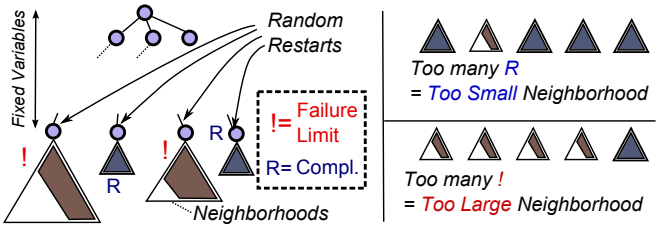

Figure 10: Tuning Large Neighborhood Search

proving optimality is intractable. In such a case, our goal is to find a high quality solution quickly.

The simplest approach is to keep running exact search algorithms until some time limit and report the best solution. In fact, this is the standard method in MIP. However, such an approach is often impractical to find good solutions within a short time period as described in Section 3.4. On the other hand, the probability of finding a good solution with a simple random sampling is too small for large problems because of the factorial number of possible orderings. One of the advantages of CP is that a CP formulation can be effortlessly extended to ***Local Search*** which addresses these problems.

Local search is a family of algorithms for quickly finding high quality solutions. There are many possible local search meta-heuristics to choose from such as, Tabu Search (TS) [12], Simulated Annealing, Ant Colony optimization, Large Neighborhood Search (LNS) [23], and Variable Neighborhood Search (VNS). We consider two TS methods, LNS and VNS. TS is a natural choice because it is effective on problems with a highly connected neighborhood (such as this one, where nearly all index permutations are feasible). We also consider LNS and VNS because they are a simple extension of a CP formulation and the CP formulation proved to be very effective on smaller instance sizes.

7.1 Tabu Search (TS)

Tabu Search (TS) is a simple method for performing gradient descent on the index permutation. At each step, TS considers swapping a pair of elements in T. To avoid being trapped in local optima and repeating the same swap, TS maintains a *Tabu list*. The elements recently swapped are considered in probation for some number of steps (called *Tabu length*). During those steps, TS does not consider swapping those elements and hopefully escapes local optima.

We implemented and evaluated two Tabu Search methods; TS-**B**Swap (*Best-Swap*) and TS-**F**Swap (*First-Swap*). TS-BSwap considers swapping all possible pairs of indexes at each iteration except the Tabu list, and takes the pair with the greatest improvement. TS-FSwap stops considering swaps when it finds the first pair that brings some improvement.

TS-BSwap will result in better quality while TS-FSwap will be more scalable because quadratic time of checking all pairs may take considerable time in large problems.

7.2 Large Neighborhood Search (LNS)

Figure 10 illustrates how a LNS algorithm executes. A LNS algorithm works by taking a feasible solution to an optimization problem and relaxing some of the decision variables. A CP search is then executed on the relaxed variables while the other variables remain fixed. If the CP search is able to assign the relaxed variables and improve the objective value, then it becomes the new current solution, otherwise the solution is reset and a new set of variables are randomly selected for relaxation (*restart*). Like most local search algorithms, this procedure is repeated until a time limit is reached. In this way, LNS leverages the power of a CP solver to efficiently search a large neighborhood of moves from the current best solution.

The CP model for our LNS algorithm was presented in Section 6.1, to complete the picture we need to explain our relaxation strategy. For simplicity we use a very basic relaxation, 5% of the indexes are selected uniformly at random for relaxation. A new relaxation is made if one of these two conditions is met; (1) the CP solver proves no better solution exists in this relaxation; (2) the CP solver has to back track over 500 times during the search (in LNS this is called the failure limit). We found this relaxation size and failure limit effectively drove the search to a high quality solution.

7.3 Variable Neighborhood Search (VNS)

One difficulty of a LNS algorithm is how to set the parameters for relaxation size and failure limit. As depicted in Figure 10, if they are set too small it is easy to get stuck in a local minimum. If they are too large the performance may degrade to a normal CP approach. Furthermore, different problem sizes may prefer different parameter settings. Our remedy for this difficulty is to change the parameters during search. This technique is well known as Variable Neighborhood Search (VNS) [11].

Our VNS approach is to start the search on a small neighborhood and inspect the behavior of the CP solver to increase the neighborhood and escape local minima only when it is necessary. The intuition is, if the relaxation terminates because the CP solver proves there is no better solution, then we are stuck in a local minimum and the relaxation size must increase. However, if the CP solver hits the failure limit without proof, then we should do more exploration in the same size neighborhood, which is achieved by increasing the failure limit. Specifically, we group the relaxations into groups of 20 and if more than 75% of these relaxations were proofs then we increase the relaxation size by 1%, otherwise we increase the failure limit by 20%.

In the experimental section, we find this VNS strategy has two benefits. First it guides the algorithm to high-quality solutions faster than a regular LNS and also consistently found higher quality solutions. Second, VNS is highly scalable and stable even for a problem with hundreds of indexes, which is not the case with the other methods.

7.4 Greedy Initial Solution

As described in the introduction, greedy algorithms are scalable but have no quality guarantees. Nonetheless, a greedy algorithm can provide a great initial solution to start a local search algorithm.

To that end, we devise a greedy algorithm which gives a much better initial solution than starting from a random permutation. The key idea of the algorithm is to consider interactions of each index as future opportunities to enable a beneficial query plan that requires two or more indexes. We greedily choose the index with the highest density (benefit divided by the cost to create the index) at each step. Here, the benefit is the query speed-up achieved by adding the index *plus* the potential benefits from interactions. We find query plans that contain the index but are not yet feasible because of missing indexes, then equally attribute the speed-up of the query plan to the missing indexes, dividing the benefit by the count of them. For more details and analysis of its quality, see the extended version [15].

8. EXPERIMENTS

We implemented our prototype of the index ordering problem

solver with a popular commercial DBMS and its design tool for the experiments. We also used COMET 2.1 as a CP/LNS solver and ILOG CPlex 12.2 as a MIP solver. All experiments are done in a single machine with a Dual-Core CPU and 2 GB of RAM. CPlex automatically parallelized the MIP on the dual core while CP and local search in COMET only used one core.

We use two standard benchmarks as datasets; TPC-H and TPC-DS. Table 4 shows the size of each dataset. TPC-DS is a major revision of TPC-H to reflect the complex query workloads and table scheme in real data analysis applications. TPC-DS has many more queries, each of which is substantially more complex and requires several indexes to efficiently process when compared to TPC-H. Hence, the design tool suggested 148 indexes (up to 300 depending on configurations of the tool). There is even a query plan that uses as many as 13 indexes together. We also found a rich set of index interactions in both datasets.

We detect various query plans and interactions as follows. We first call the DBMS's what-if query optimizer with all hypothetical indexes suggested by the DBMS's database designer. The query optimizer returns the best *atomic configuration* [10]. We then remove the hypothetical indexes in the atomic configuration and call the optimizer again, getting a sub-optimal atomic configuration. We repeat these steps several times for each query. The resulting set of atomic configurations are the query plans, from which we extract the competing and query interactions. We do the same with the queries to create indexes for detecting the build interactions.

Table 4: Experimental Datasets

Dataset	$\|Q\|$	$\|I\|$	$\|P\|$	Largest Plan	#Inter. (Build)	#Inter. (Query)
TPC-H	22	31	221	5 Index	31	80
TPC-DS	102	148	3386	13 Index	243	1363

8.1 Exact Search Results

We verified the performance of each method to find and prove the optimal solution with the TPC-H dataset.

We compared the performance of MIP and CP methods with and without the additional constraints, varying the number of indexes (size of the problem). For MIP, we discretized the problem for $|I| * 20$ timesteps. We also varied the density of the problem. *low* density means we remove all suboptimal query plans and build interactions. *mid* density means we remove all but one suboptimal query plan and build interactions with less than 15% effects.

As can be seen in Table 5, neither MIP nor CP could solve even small problems without problem specific constraints, taking time that grows factorially with the number of indexes. By applying the problem specific constraints (denoted by $^+$), both MIP and CP were dramatically improved and took less than one minute to solve all low-density problems. For higher density problems, they took substantially longer because the pruning power of additional constraints decreases. MIP suffered more from the higher density because it results in more non-linear properties discussed in Section 6. VNS quickly found the optimal solution in all cases. In the 21 indexes and mid-density problem, VNS found a good solution within one minute and did not improve the solution for 3 hours. This strongly implies the solution is optimal, but there is no proof as the exact search methods did not finish.

Drill-Down Analysis: Table 6 shows how the additional constraints from each problem property affects the performance of the complete search experiment described in Section 8.1. We start with no additional constraint and add each problem property one at a time in the following order, **A**lliances, **C**olonized-indexes, **M**in/max-domination, **D**isjoint-clusters, and **T**ail-indexes. We only used additional constraints we could deduce within one minute, so the overhead of pre-analysis is negligible.

Table 5: Exact Search (Reduced TPC-H): Time [min]. Varied the number and interaction density of indexes. VNS: No optimality proof. DF: Did not Finish in 12 hours or out-of-memory.

$\|I\|$ Density	6 low	11 low	13 low	22 low	31 low	16 mid	21 mid
MIP	<1	11	106	DF	DF	DF	DF
CP	<1	7	214	DF	DF	DF	DF
MIP$^+$	<1					168	DF
CP$^+$	<1					1	DF
VNS	<1						<1?

Table 6: Pruning Power Drill-Down (Reduced TPC-H). Time [min].

$\|I\|$ Density	6 low	11 low	13 low	18 low	22 low	25 low	31 low	16 mid	21 mid
CP	<1	7	214	DF	DF	DF	DF	DF	DF
+A	<1			DF	DF	DF	DF	DF	DF
+AC	<1			69	DF	DF	DF	DF	DF
+ACM	<1				249	DF	DF	DF	DF
+ACMD	<1					24	DF	DF	DF
+ACMDT	<1							1	DF

Table 7: Greedy, Dynamic Programming, and 100 Random Permutations for Initial Solutions. (TPC-DS is 400 times larger in scale.)

Dataset	Greedy	DP	Random (AVG)	Random (MIN)
TPC-H	47.9	57.0	65.5	51.5
TPC-DS	65.9	70.5	74.1	69.6

The results demonstrate that each of the five techniques improves the performance of the CP search by several orders of magnitude without affecting optimality. The runtime of CP without pruning is roughly proportional to $|I|!$. Hence, the total speed-up of the additional constraints is at least $\frac{31!}{13!}214 = 2.7 \times 10^{26}$.

8.2 Local Search Results

We also studied TPC-H and TPC-DS with all indexes, query plans, and interactions. Because of the dense interactions and many more indexes, the search space increases considerably. Even CP with the problem specific constraints cannot prove optimality for this problem and gets stuck in low quality solutions. Hence, we used our local search algorithms to understand how to find high quality solutions to these large problems.

Limited Scalability of Exact Search: The MIP model suffers severely on these large problems and CPlex quickly runs out of memory before finding a feasible solution with as much as 4 GB of RAM. This is because the denser problem significantly increases the number of non-zero constraints and variables, and CPlex cannot significantly reduce the problem size in the pre-solving step. In fact, over 1 million integer variables remain after pre-solving for problems of this size. This result verifies that a linear system approach does not scale well for the index ordering problem.

Although we also tested CP in this and next experiment, CP takes a long time to find a solution better than the initial greedy solution because it is overwhelmed by a large neighborhood. These results

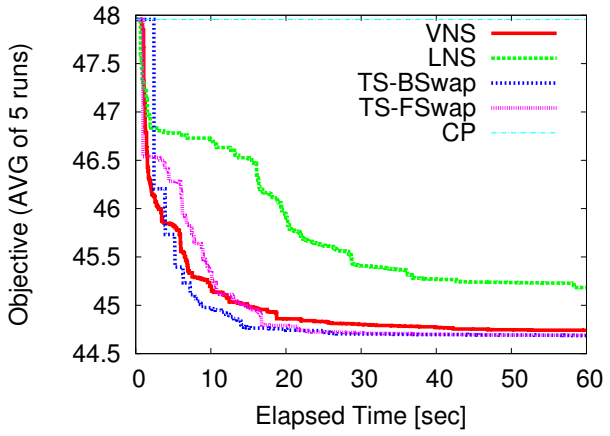

Figure 11: Local Search (TPC-H): LNS, VNS and Tabu. (MIP runs out memory)

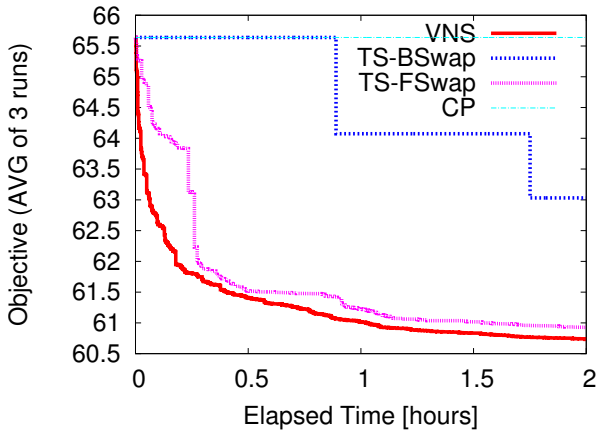

Figure 12: Local Search (TPC-DS): VNS and Tabu. (MIP runs out memory)

demonstrate the need for local search methods in larger problems as described in Section 7.

We then evaluated the performance of local search algorithms (TS, LNS, and VNS) described in Section 7 on these problems. All the local search methods are implemented in COMET and given the same constraints with the same initial solution.

Algorithm Comparison for Initial Solution: Our local search uses the greedy algorithm described in Section 7.4 to come up with the initial solution. We compared the quality of the initial solution with a Dynamic Programming (DP) algorithm suggested earlier by Schnaitter et al [21]. Detailed algorithm of our greedy and our implementation of the DP algorithm is given in the extended version of the paper [15].

Table 7 shows the objective value of the solutions suggested by our greedy, DP, and the average and minimum values of 100 random permutations of indexes. Our greedy solutions are always better than both the average and minimum of random permutations as well as than the DP algorithm.

The main reason our greedy algorithm achieves the better quality than the DP algorithm is that the DP algorithm does not consider how long building each index will take, assuming all index creation costs are uniform. Hence, it often chooses a compact index later even if the index has high *density* (benefit divided by creation cost).

Another problem in both our greedy and DP is that they do not consider build interaction to speed-up deployment time. The resulting index orders often do not have fast deployment time, which is one reason we need to improve the initial solution by the local search.

TPC-H Results: Figure 11 shows the quality (y-axis) of solutions plotted against elapsed search time (x-axis) for the TPC-H dataset. The figure compares the LNS, VNS and two Tabu Search (TS) methods described in Section 7.

In this experiment, TS-BSwap achieves a better improvement than TS-FSwap because TS-BSwap considers all possible swaps in each iteration. VNS is comparable to the two Tabu methods while the original form of LNS takes a long time to improve the solution because it cannot dynamically adjust the size of its neighborhood. We also observed that VNS is more *stable* than LNS in that it has less variance of solution quality between runs.

TPC-DS Results: Figure 12 compares VNS with Tabu Search for the TPC-DS dataset. This time, the improvement of TS-BSwap is large but very slow because it takes a very long time (50 minutes) for each iteration to evaluate $\binom{148}{2}$ swaps. VNS achieves the best improvement over all time ranges, followed by TS-FSwap. VNS quickly improves the solution, especially at the first 15 minutes.

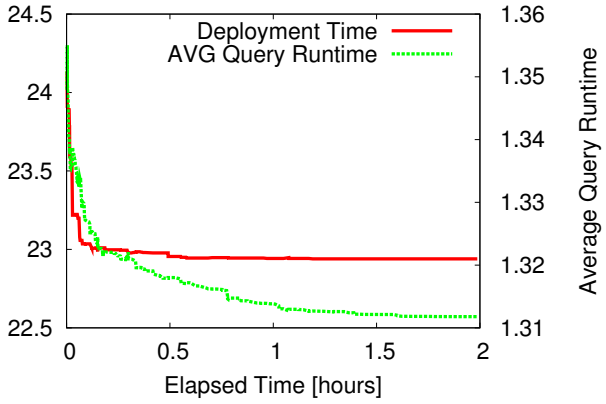

Figure 13: VNS (TPC-DS): Deployment Time and Average Query Runtime.

Considering that deploying the 148 indexes on the Scale-100 instance takes one day, VNS achieves a high quality solution within a reasonable analysis time.

Figure 13 plots the index deployment time and average query runtime during the deployment period to analyze where the improvements of VNS comes from at each time range. The sharp improvement at the beginning (15 minutes) of Figure 12 is mainly attributed to the improvement on deployment times by exploiting build interactions between indexes. After that, VNS mainly improves the average query runtime by deploying a set of indexes that have significant speed-ups at early steps.

8.3 Discussions

Scalability and Robustness: The result shows that VNS is a scalable and robust local search method which quickly finds high quality solutions in all cases tested. The main reason the TS methods sometimes do not work well is essentially the same as why the LNS with fixed parameters does not perform well. The neighborhood size is fixed and it may be too large with TS-BSwap or too small with TS-FSwap.

It is possible to devise a hybrid Tabu method that dynamically adjusts the tuning parameters (the number of pairs to check, Tabu length, etc) for the problem, but VNS has another important property for avoiding local optima. As VNS relaxes more than two variables at each iteration, it can explore multi-swap neighborhoods that are necessary to influence large sets of interacting indexes.

Applicability to Database Design Tools: Because of the scalability and robustness, VNS on top of CP formulation is highly promising to physical database design problems in general such as index selection.

Although the database community has made several efforts towards MIP and BIP (Boolean Integer Programming) for physical database design tools [9, 17, 19], none of commercial tools has employed those methods so far.

One of the vendors told the authors that the main reason to stick with greedy algorithm is its scalability for substantially large and complex query workloads in the real world up to millions of distinct queries. As a commercial tool, it is unacceptable even for such huge problems to expose too long runtime (e.g., days to suggest the *first* design) or too unstable quality (e.g., missing indexes that are crucially important) when terminated earlier.

Unlike integer programming, CP formulation achieves the scalability with robust solution quality by starting from greedy algorithm and quickly improving it with VNS. Hence, we consider CP and local search as the primary approach for our next step towards a database design tool that incrementally optimizes databases.

9. CONCLUSION AND FUTURE WORK

In this paper, we proposed our vision towards a physical database design tool for large databases to accommodate frequent and drastic changes in query workloads, logical and physical table schema. We call our new design approach as Incremental Database Design which differs from both the traditional off-line design tools and online index selection approaches. The key requirements is to minimize administrative costs to repeatedly tune large data-warehouses without sacrificing query performance improvements.

As the first step, we defined and solved the optimization problem of index deployment ordering. We formalized the problem using a mathematical model and studied several problem specific properties which increase performance of industrial optimization tools by several orders of magnitude. We developed several approaches for solving the problem including, a greedy algorithm, CP formulation, MIP formulation, and four local search methods. We demonstrated that this problem is best solved by a CP framework and found that our VNS local search method is robust, scalable, and quickly finds high quality solutions on very large problems.

Our next step is to jointly solve the index selection problem and index deployment ordering problem. We are currently working on an integrated solution that accounts for the index deployment ordering while choosing a set of indexes to build. The main challenges are two fold. First, as we described in this paper, scheduling an optimal deployment order for a single given set of indexes is already an expensive analysis. It is obviously impractical to consider the order of indexes for every candidate design. Second, now that we include in our design tool the deployment time and how quickly users will see the query speed-up, we need to provide flexible yet easy-to-understand interfaces to let DBAs state their requirements in this multi-objective optimization problem. We will tackle these issues based on our prior physical database design tool [17] and our scalable CP/VNS optimization methods developed in this paper.

Acknowledgment

We would like to thank Harumi Kuno at Hewlett-Packard Laboratories for her invaluable helps to edit our paper. This work is supported by the NSF under the grant IIS-0916691.

10. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[2] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, 2006.

[3] P. Baptiste, C. L. Pape, and W. Nuijten. *Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems*. Springer, 2001.

[4] M. Blaschka, C. Sapia, and G. Hofling. On schema evolution in multidimensional databases. *DataWarehousing and Knowledge Discovery*, pages 153–164, 1999.

[5] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[6] N. Bruno and S. Chaudhuri. Physical design refinement: The merge-reduce approach. *TODS*, 2007.

[7] S. Chaudhuri and V. Narasayya. AutoAdmin what-if index analysis utility. In *SIGMOD*, 1998.

[8] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.

[9] D. Dash, N. Polyzotis, and A. Ailamaki. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. In *PVLDB*, volume 4, 2011.

[10] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *TODS*, 13(1):91–128, 1988.

[11] F. Glover and G. Kochenberger. *Handbook of metaheuristics*. Springer, 2003.

[12] F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.

[13] H. Gupta and I. Mumick. Selection of views to materialize under a maintenance cost constraint. *ICDT*, 1999.

[14] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[15] H. Kimura, C. Coffrin, A. Rasin, and S. Zdonik. A Constraint-Programming Approach To Optimize Index Deployment Order (Extended Version). *Arxiv/CoRR abs arXiv:1107.3606*, 2011.

[16] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. In *VLDB*, 2009.

[17] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. CORADD: Correlation aware database designer for materialized views and indexes. *VLDB*, 2010.

[18] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, 1992.

[19] S. Papadomanolakis and A. Ailamaki. An Integer Linear Programming Approach to Database Design. In *ICDE*, 2007.

[20] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-Line Index Selection for Shifting Workloads. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, 2007.

[21] K. Schnaitter et al. Index interactions in physical design tuning: Modeling, analysis, and applications. *VLDB*, 2009.

[22] J. P. Sousa and L. A. Wolsey. A time indexed formulation of non-preemptive single machine scheduling problems. *Mathematical Programming*, 54:353–367, 1992.

[23] P. Van Hentenryck and L. Michel. *Constraint-based local search*. The MIT Press, 2009.

[24] D. C. Zilio et al. DB2 design advisor: integrated automatic physical database design. In *VLDB*, 2004.