## Optimizing Index Deployment Order for Evolving OLAP (Extended Version)

Hideaki Kimura  Carleton Coffrin  Alexander Rasin  Stanley B. Zdonik

\{hkimura, cjc, alexr, sbz\}@cs.brown.edu, Brown University, Providence, RI

**Abstract**—Query workloads and database schemas in OLAP applications are becoming increasingly complex. Moreover, the queries and the schemas have to continually evolve to address business requirements. During such repetitive transitions, the order of index deployment has to be considered while designing the physical schemas such as indexes and MVs.

An effective index deployment ordering can produce (1) a prompt query runtime improvement and (2) a reduced total deployment time. Both of these are essential qualities of design tools for quickly evolving databases, but optimizing the problem is challenging because of complex index interactions and a factorial number of possible solutions.

We formulate the problem in a mathematical model and study several techniques for solving the index ordering problem. We demonstrate that Constraint Programming (CP) is a more flexible and efficient platform to solve the problem than other methods such as mixed integer programming and A* search. In addition to exact search techniques, we also studied local search algorithms to find near optimal solution very quickly.

Our empirical analysis on the TPC-H dataset shows that our pruning techniques can reduce the size of the search space by tens of 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 a near-optimal solution.

## I. INTRODUCTION

The selection and deployment of indexes is been one of the most important, and also one of the most challenging tasks of database administrators (DBAs). Both industry and academia have intensively focused their study on the automatic selection of indexes in physical database design [1], [2]. Every modern commercial database management system (DBMS) ships an automatic design tool as one of its key components. These design tools support DBAs by suggesting sets of indexes that dramatically improve query execution.

However, recent applications require complex data processing over hundreds or thousands of tables [3], making the selection of an appropriate set of indexes impossible for human DBAs and extremely challenging for automated tools. Moreover, the queries, the data, and even the schema in very large data-warehouses are continually evolving [4]. Several iterations are often required to accurately translate business requirements into a database schema and these requirements will change over time.

For example, imagine a popular online digital music shop, iTunes 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 accounts to buy music from localized versions of iTunes Store in different countries. Thus, the company decided to tie each customer to multiple countries. To accommodate this small change, the logical database schema evolved to add a new `n:n` table `CUST_COUNTRIES(CUSTID, COUNTRY)`, eliminating the `COUNTRY` column from the `CUSTOMER` table.

Such relatively minor change in the schema will have a cascading effect on the physical design used for analysts’ reports. Every index and view that used the `COUNTRY` column has to be rebuilt and a new materialized view that joins `COUNTRY` and `CUST_COUNTRIES` is needed or substitute indexes have to be built instead. In short, every physical structure that contained the `COUNTRY` column must be dropped and rebuilt, which means this single column alteration can affect dozens of indexes. Thus a change in a few of the “popular” columns can require replacing the majority of the indexes in the current design, not to mention the brand new indexes that may need to be added.

One emerging approach for solving this problem is the online-index selection [5], [6]. The main idea is to monitor the database queries and to deploy (or drop) indexes when it sees a shift in the query workload. The online approach can quickly react to changes in the database or the workload. The idea is that a 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 handling 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 a local optima. This problem is particularly difficult when a change of both the query workload and the logical table schema is needed. Even a small change in business requirements can require drastically different queries as well as logical and physical design.

The online index approach cannot capture the impact of
such a change. The common approach in online index selection is to pre-compute a set of potentially beneficial indexes and only re-evaluate their benefits for queries monitored at runtime [5], [6]. 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. Thus a small number of new complex analytic queries in the workload can require tens of additional indexes. The underlying interactions that speed-up index creation require optimizing the order of index deployment studied in this paper. Exploiting the complex index interactions requires a detailed analysis involving millions of queries and thousands of candidate indexes, which is impractical for on-line design tools.

All of these problems are rooted in the assumption that because the index selection and deployment process takes place entirely online, the selection algorithm must be as low-overhead as possible in order to continuously monitor query workloads and quickly react to workload shifts.

A. Incremental Database Design

Motivated by the observations above, we have begun exploring a new approach, which, as illustrated in Figure 1, is positioned between the two extremes of completely off-line and completely on-line. Our target is a very large data-warehouse which needs a significant change in its physical design. In this case, it is necessary to consider both the query runtime and the deployment time of the suggested indexes to incrementally evolve the running system. At the same time, we assume that change is relatively infrequent (e.g., a week) compared to what on-line index approach assumes (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 an Incremental Database Design (IDD) and are studying its requirements, design and implementation as a long term project.

The first challenge, and the focus of the current paper, is scheduling the deployment of indexes to minimize the index deployment time and speed-up the majority of queries quickly. 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. For instance, deploying one index over a table that stores billions of tuples (which is not uncommon) could take days.

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

B. 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 benefit as illustrated in Figure 2. First, a good order achieves prompt query runtime improvements by deploying indexes that yield the greatest query speed-up at the early stages. 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 [7], both the benefit and the build cost of an index are dependent on the previously built indexes because of index interactions described in Section III-B. Certain 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 use a greedy heuristic to quickly choose a solution. However, the quality of greedy approaches 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.

C. Paper Overview

In this paper, we formally define the ordering problem as a mathematical model and propose several pruning techniques based not on linear relaxation but rather on the combinatorial properties of the problem. We show that these problem-specific combinatorial properties can reduce the size of the search space by tens of 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
a near-optimal solution 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. Finally, we discuss our next step towards incremental database design based on this work to optimize index deployment order.

In summary, our contributions are: (1) Framework of incremental database design, (2) A formal description of the index deployment order problem, (3) Problem-specific properties to reduce problem difficulty, (4) Models and algorithms for Greedy, MIP, CP and local search (5) Analysis of various solution techniques and solvers, (6) Empirical analysis on TPC-H and TPC-DS. We note that, 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.

II. RELATED WORK

A. 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 [2], [8].

The AutoAdmin project [1] pioneered this field. It creates a set of hypothetical, potentially beneficial indexes, and uses the what-if interface of the optimizer [9] to evaluate their expected benefit. Once we have an estimate for the benefit of each index, 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 [2] (benefit/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 A* search [10], mixed integer programming (MIP) [11], [12] and boolean integer programming (BIP) [13].

Despite the wealth of research in physical database design, the problem of optimizing the 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 [14] which considers a query workload as a sequence, but only considers dropping and re-creating existing indexes to reduce maintenance overhead. The work in [15] also considered ordered deployment, but primarily as a way to greedily speed up queries at every step rather than to optimize the overall index deployment sequence. Bruno et al. [16] mentioned a type of ordering problem as an unsolved problem, but their objective does not consider prompt query speed-ups. Also, they only suggested using A* or Dynamic Programming and did not solve the problem in [16].

B. Online Index Selection

Schnaitter et al. proposed the COLT framework [6] 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 a time, 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 [15] 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 III-A. However, their solution to the 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, their approach introduces a number of simplifications, such as assuming the cost to build each index is uniform and unchanged by index interaction.

Although some have explored the problem of index benefit interaction [5], [15], 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 an 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 III-B such interaction can have a significant effect on the overall index deployment cost. In this work we use the 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 III-B) to find a good index deployment order.
TABLE I
MIP AND CP COMPARISON

<table>
<thead>
<tr>
<th>Constraints &amp; Objectives</th>
<th>MIP</th>
<th>CP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear &amp; Non-Linear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pruning Method</td>
<td>Branch-Bound &amp; Linear Relaxation</td>
<td>Branch-Prune &amp; Custom Constraints</td>
</tr>
<tr>
<td>Non-Exhaustive Search Variant</td>
<td>N/A</td>
<td>Local Search</td>
</tr>
<tr>
<td>Best Suited for</td>
<td>Linear Problems</td>
<td>Combinatorial Problems</td>
</tr>
</tbody>
</table>

C. Branch-and-Bound

All decision problems, such as the index order problem, can be formulated as a tree search problem. 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 (such as, the traveling salesman problem). MIP only supports linear constraints, and thus it is hard to model non-linear properties in MIP.

D. 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 I.

First, CP uses a branch and prune 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 [7].

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 subtree that has some chance of containing the optimal solution regardless of how slight the chance is, and how large the subtree 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) [17] 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.

III. 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 II and III.

A. 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 \( i_3 \to i_1 \to i_2 \). 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 illustrated in Figure 3.

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.

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

B. 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.

TABLE II
SYMBOLS & CONSTANT VALUES (IN LOWER LETTERS)

| i ∈ I | An index. \( I = \{i_1, i_2, ..., i_P \} \) |
| q ∈ Q | A query. |
| p ∈ P | A query plan (a set of indexes). |
| plans(q) ∈ P | Feasible query plans for query \( q \). |
| ctime(q) | Original runtime of query \( q \). |
| qspeedup(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 \( i \). |
| cspeedup(i, j) | Speed-up of using index \( j \) for index \( i \). |
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(Sal) FROM People WHERE City=Prov
```

Assume the query plan using $i_1$ is 5 seconds faster than a full scan while 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.

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. [16] also mentioned this effect earlier. In Section VII we show there exist a rich set of such interactions.

**Precedence:** Some index must precede another index.

One example is an index on a materialized view (MV). A MV 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.

Another example is a secondary index that exploits correlation [18]. 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 [15]. In our experiments, we detect interactions by calling the query optimizer with hypothetical indexes as detailed in Section VII.

### C. Mathematical Model

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

**Objective:**

\[
\min \sum_i (R_{i-1}C_i) \quad (1)
\]

**Subject to:**

\[
Y_{p,i} = \{T_j \leq i : \forall j \in p\} : \forall p,i \quad (2)
\]

\[
X_{q,i} = \max_{p \in \text{plans}(q)} qspdup(p,q)Y_{p,i} : \forall q,i \quad (3)
\]

\[
R_i = \sum_q (qtime_q - X_{q,i}) : \forall i \quad (4)
\]

\[
C_T = ctime(i) - \max_{j:T_j < T_i} cspdup(i,j) : \forall i \quad (5)
\]

Constraint 2 states that a query plan is available only when all of the indexes in the query plan are available. Constraint 3 calculates the query speed-up by using the fastest query plan for the query at a given time. Constraint 4 sums up the speed-ups of each query and subtract from the original query runtime to get the current total runtime.

Constraint 5 calculates the cost to create index $i$ ($C_T$ 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 observed 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 a near-optimal solution that can be found quickly.

### D. Discussion

There could be variants of the objective. For example, putting different weights on particular queries can be incorporated by simply scaling up or down runtimes of the queries. Or, one can simply consider $\sum C_i$ as objective to minimize the deployment time like [16]. In either case, most of the modeling and pruning strategies in this paper will be usable with minor modifications.

### IV. Problem Properties

This problem has up to $|I|!$ possible solutions. An exhaustive search method that tests all the solutions becomes intractable even for small problems. Hence, 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 Appendix IV.

These are inherent problem properties 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 VII.

A. Alliances

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

Figure 4 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.

B. 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 5 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 never yields a 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.

C. 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 Appendix IV.

Figure 6 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 \(i_3\) occurs when there already exists \(i_4\), 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 \(i_1\); \(T_{i_1} > T_{i_2}\).

D. Disjoint Indexes and Clusters

The next problem property is called a disjoint index and 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 7 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 indexes other than themselves.

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 7 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 \(den_{i_1+i_2} > den_{i_4i_2} \leq den_{i_4i_2} > den_{i_4i_3}\) and \(den_{i_4i_2} > den_{i_4i_3}\) 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 considering a subsequence \( i \to X \to j \) for arbitrary \( X \), so \( i \) is disjoint in a backwards order. Because of the property of disjoint indexes, the subsequence must satisfy \( \text{den}_i < \text{den}_j \) if it is optimal. Thus, if we know \( \text{den}_i > \text{den}_j \), we can prune out all solutions that build \( j \) before \( i \). Conversely, if \( i \) is forward-disjoint and \( \text{den}_i < \text{den}_j \), then \( i \) always succeeds \( j \).

### E. Tail Indexes

Because of the inequality constraints given by the above properties, sometimes a single index is uniquely determined to be the last 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 VII-C, \( i_2 \) 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 8. 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 calculate the tail objectives because the set of preceding indexes is known and, 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 \( i_2 \to i_5 \to i_1 \) in Figure 8 is smaller than that of \( i_4 \to i_1 \to i_2 \). 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. \( i_4 \to i_1 \to i_2 \) and \( i_1 \to i_4 \to i_2 \) 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 8 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.

### F. 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-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.

### V. Constraint Programming

In this section, we describe how we translate the mathematical model given in Section III-C 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.

#### A. CP Model

CP allows a flexible model containing both linear and nonlinear objectives and constraints. The mathematical formulation presented in Section III-C can be modeled in standard CP.
<table>
<thead>
<tr>
<th>Tail</th>
<th>Obj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>i₄ → i₁ → i₂</td>
<td>9.7</td>
</tr>
<tr>
<td>i₄ → i₂ → i₁</td>
<td>9.9</td>
</tr>
<tr>
<td>i₁ → i₄ → i₂</td>
<td>12</td>
</tr>
<tr>
<td>i₅ → i₁ → i₂</td>
<td>4.0</td>
</tr>
<tr>
<td>i₅ → i₂ → i₁</td>
<td>4.2</td>
</tr>
<tr>
<td>i₂ → i₅ → i₁</td>
<td>4.5</td>
</tr>
<tr>
<td>i₈ → i₁ → i₂</td>
<td>6.8</td>
</tr>
<tr>
<td>i₈ → i₂ → i₁</td>
<td>6.9</td>
</tr>
<tr>
<td>i₁₁ → i₁ → i₂</td>
<td>7.1</td>
</tr>
<tr>
<td>i₁₁ → i₂ → i₁</td>
<td>7.3</td>
</tr>
</tbody>
</table>

**Fig. 8.** Comparing Tail Indexes of Same Index Set in TPC-H

solvers (e.g., COMET) almost identically unlike MIP where the model is more obfuscated (an equivalent MIP model is given in Appendix [II]).

**Objective:** $\min \sum_i (R[i] - 1)C[i]$ (6)

Subject to:

| $\text{allDifferent}(T)$ | $\forall p, i$ |

$Y[p, i] = \bigwedge_{j \in p} (T[j] \leq i): \forall p, i$ (7)

$X[q, i] = \max_{p \in \text{plans}(q)} (qspdup(p, q, Y[p, i])) : \forall q, i$ (8)

$R[i] = \sum_q (qtime(q) - X[q, i]) : \forall i$ (9)

$C[T[i]] = ctime(i) - \max_{j}(T[j] < T[i])cspdup(i, j) : \forall i$ (10)

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

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

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 [19].

However, in addition to losing the 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 (10) which uses allDifferent. This interesting constraint in CP assures all 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 (3). 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 as 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 $C_T$. 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 all of the constraints and the objective calculation.

**Additional constraints:** Finally, we add the additional constraints developed in Section [IV] to reduce the search space.

**B. 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-prune 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 [VIA]. It also utilizes the problem specific constraints we developed in Section [IV] to efficiently explore only high quality 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.

**VI. LOCAL SEARCH**

Although CP is well suited for this ordering problem, when there is a large number of indexes with dense interactions between them, proving optimality is intractable. In such a case, our goal is to find a near-optimal 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 I-D. One of the goals to use CP is that a CP formulation can be effortlessly extended to Local Search which addresses this problem.

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) [20], Simulated Annealing, Ant Colony optimization, Large Neighborhood Search (LNS) [17], 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.

A. 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-BSwap (Best-Swap) and TS-FSwap (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.

B. Large Neighborhood Search (LNS)

Figure 9 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 V-A 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.

C. 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 9 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) [21].

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.

Fig. 9. Tuning Large Neighborhood Search

Fig. 10. System Overview


D. 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 usable 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 Appendix III.

VII. Experiments

A. Implementation

Our implementation of the CP-based index ordering solution is summarized in Figure 10. Given a query workload, we first run a physical database design tool to obtain a set of suggested indexes. To avoid actually creating indexes, we use the what-if [9] interface of the DBMS which will hypothetically create each index and estimate the cost of a query in the query optimizer’s cost model. 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 apply the optimization techniques described in Section IV to get additional constraints to speed up the CP solver. The CP/LNS engine then produces the optimized index deployment order.

We used a popular commercial DBMS and its design tool for the experiments. We 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.

B. Datasets

We use two standard benchmarks as datasets; TPC-H and TPC-DS. Table IV shows the size of each dataset. TPC-DS is a major revision of TPC-H that reflects the complex query workloads and table schema in real data analysis applications. TPC-DS has many more queries, each of which is substantially more complex and can benefit from several indexes in contrast to TPC-H. Hence, the design tool suggested 148 indexes (up to 300 depending on the configuration of the tool). One query plan uses 13 indexes. We also found a rich set of index interactions in both datasets.

C. Exact Search Results

We compared the performance of MIP and CP methods on the TPC-H dataset with and without the additional constraints, varying the number of indexes (size of the problem). For MIP, we discretized the problem for $|I| \times 20$ 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.

Table V shows that 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 in the higher density case because it results in more non-linear properties as discussed in Section VII-C. 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 by running 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 VI shows how the additional constraints from each problem property affects the performance of the complete search experiment described in Section VII-C. We start with no additional constraint and add each problem property one at a time in the following order, Alliances, Colonized-indexes, Min/max-domination, Disjoint-clusters, and Tail-indexes. We only used additional constraints we could derive within one minute, so the overhead of pre-analysis is negligible.

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 $\frac{31!}{13!} \times \frac{214}{6} = 2.7 \times 10^{26}$.

D. 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 achieve 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 in 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
TABLE V
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.

<table>
<thead>
<tr>
<th>Density</th>
<th>6</th>
<th>11</th>
<th>13</th>
<th>22</th>
<th>31</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIP</td>
<td>&lt;1</td>
<td>11</td>
<td>106</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
<tr>
<td>CP</td>
<td>&lt;1</td>
<td>7</td>
<td>214</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
<tr>
<td>MIP+</td>
<td>&lt;1</td>
<td>168</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
<tr>
<td>CP+</td>
<td>&lt;1</td>
<td>1</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
<tr>
<td>VNS</td>
<td>&lt;1</td>
<td>17</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
</tbody>
</table>

TABLE VI
PRUNING POWER DRILL-DOWN (REDUCED TPC-H): TIME [MIN].

<table>
<thead>
<tr>
<th>Density</th>
<th>6</th>
<th>11</th>
<th>13</th>
<th>18</th>
<th>22</th>
<th>25</th>
<th>31</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP+</td>
<td>&lt;1</td>
<td>7</td>
<td>214</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
<tr>
<td>+A</td>
<td>&lt;1</td>
<td>69</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
<tr>
<td>+AC</td>
<td>&lt;1</td>
<td>249</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
<tr>
<td>+ACMD</td>
<td>&lt;1</td>
<td>1</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
<tr>
<td>+ACMDT</td>
<td>&lt;1</td>
<td>1</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
<td>DF</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Greedy</th>
<th>DP</th>
<th>Random (AVG)</th>
<th>Random (MIN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-H</td>
<td>47.9</td>
<td>57.0</td>
<td>65.5</td>
<td>51.5</td>
</tr>
<tr>
<td>TPC-DS</td>
<td>65.9</td>
<td>70.5</td>
<td>74.1</td>
<td>69.6</td>
</tr>
</tbody>
</table>

TPC-H dataset. The figure compares the LNS, VNS and two Tabu Search (TS) methods described in Section VI.

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-FSwap is large but very slow because it takes a very long time (50 minutes) for each iteration to evaluate $\binom{25}{2}$ 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 plots the index deployment time and average query runtime during the deployment period to analyze where the improvement of VNS comes from at each time range. The sharp improvement at the beginning (15 minutes) of Figures 12 is mainly attributed to the improvement in deployment times by exploiting build interactions between indexes. After that, VNS primarily improves the average query runtime by deploying a set of indexes that have significant speed-up at early steps.

E. Discussions

Scalability and Robustness: The result verifies that VNS is a scalable and robust local search method which quickly finds a high quality solution in all cases. The main reason the TS methods sometimes fail to work well is essentially the same as why 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 Physical Database Design Tools:
Because of the scalability and robustness, VNS on top of a CP formulation is a highly promising approach 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 [11]–[13], none of commercial tools has employed those methods so far.

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.

CP takes long time to find a solution better than the initial greedy solution because it is trapped in too large a neighborhood. These results demonstrate the need for local search methods in larger problems as described in Section VI.

We then evaluated the performance of local search algorithms (TS, LNS, and VNS) 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 VLD 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. [15]. Detailed algorithm of our greedy and our implementation of the DP algorithm is given in Appendix III.

Table VII 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 the DP algorithm.

The main reason our greedy algorithm achieves better quality than DP is that DP assumes 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 using the local search.

TPC-H Results: Figure 11 shows the quality (y-axis) of solutions plotted against elapsed search time (x-axis) for the
Unlikely integer programming, a CP formulation achieves a robust and scalable solution 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.

VIII. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we proposed our vision towards a physical database design tool for large databases to accomodate frequent and drastic changes in query workloads, logical and physical table schema. We call our new design approach Incremental Database Design. It falls between the traditional off-line design tools and on-line index selection approaches. The key requirement 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 a near-optimal solution 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 the multi-objective optimization problem. We plan to tackle these issues based on the scalable CP/VNS optimization methods developed in this paper.

ACKNOWLEDGMENT

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

REFERENCES

**APPENDIX I**

**SOURCE CODE, DATASETS**

All of the source code and experimental data can be accessed on the web at [http://cs.brown.edu/~cjc/idd/](http://cs.brown.edu/~cjc/idd/). This includes Java projects, CPLEX/COMET models, and problem data files.

Our purpose is two-fold. First, we would like to ensure the reproducibility of our experiments. Second, we expect this problem will be useful for testing various solver technologies and we want to make it available to the operation research community.

**APPENDIX II**

**FULL MIP MODEL**

This section provides the detailed MIP model for the index ordering problem. The model uses the input data described in Table VII and defines additional constants and variables in Table VIII. Variables annotated with a hat have slightly different semantics than those in the CP model, but their meaning is roughly the same. The biggest decision variable change is that the B variables are used to determine the orders of indexes.

<table>
<thead>
<tr>
<th>TABLE VIII</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADDITIONAL SYMBOLS &amp; VARIABLES</strong></td>
</tr>
<tr>
<td>d ∈ D</td>
</tr>
<tr>
<td>Ai ∈ D</td>
</tr>
<tr>
<td>Bi,j ∈ {0, 1}</td>
</tr>
<tr>
<td>Ci</td>
</tr>
<tr>
<td>Xq,d</td>
</tr>
<tr>
<td>Yq,p,d ∈ {0, 1}</td>
</tr>
<tr>
<td>Zi,d ∈ {0, 1}</td>
</tr>
<tr>
<td>CYi,j ∈ {0, 1}</td>
</tr>
</tbody>
</table>

The index order problem can be formulated as a MIP as follows,

**Objective:**

min \( \sum_d \left( \sum_q \hat{X}_{q,d} \right) \) \tag{12}

**Subject to:**

\( B_{i,j} \leq B_{i,k} : \forall i \neq j \) \tag{13}

\( B_{i,k} \leq B_{i,j} + B_{j,k} : \forall i \neq j \neq k \) \tag{14}

\( B_{i,j} \leq 1 - \frac{A_i + \hat{C}_i - A_j}{|D|} : \forall i \neq j \) \tag{15}

\( \sum_p \hat{Y}_{q,p,d} = 1 : \forall q, d \) \tag{16}

\( \hat{X}_{q,d} = \sum_{p \in \text{plans}(q)} \{ \hat{Y}_{q,p,d} \} \) \tag{18}

\( qtime(q) = qspdup(p, q)) : \forall q, d \) \tag{19}

\( Z_{i,d} \leq 1 - \frac{A_i + \hat{C}_i - d}{|D|} : \forall i, d \) \tag{20}

\( \sum_j CY_{i,j} \leq 1 : \forall i \) \tag{21}

\( CY_{i,j} \leq B_{j,i} : \forall i, j \) \tag{22}

\( \hat{C}_i = ctime(i) - \sum_{j \in I} (cspdup(i, j) CY_{i,j}) : \forall i \) \tag{23}

[1] assures either i precedes j or j precedes i. [14] assures the index order preserves transitivity; i cannot precede k if j precedes i and k precedes j. The A variables determine when each index is made. [15] means that, when i precedes j, A_i has to be C_i (cost to create i) smaller than A_j. A_i + \hat{C}_i - A_j is divided by |D| to normalize the expression to a range between 0 and 1.

The Y variables determine whether the plan is used for each query at d. Therefore, the sum of Y is always 1 [16]. There is always an empty-plan \{\emptyset\} which gives no speed-up to ensure feasibility of [16]. [17] assures the plan is available only when all indexes in the plan are available. Then, [19] calculates the runtime of each query from Y.

As constraints [15-17] calculate the query performance at a given time, constraints [20-23] calculate the query build cost at a given time. [20] determines whether each index is available at each time step by checking A and C. [21] and [22] are equivalent to the constraints on Y except the interaction to build index is always pair-wise. [23] calculates the time to create each index from them.

The instance specific constraints developed in Section [IV] are implemented in this model by posting constraints on A and B (e.g., i_3 < i_5 yields, B_{3,5} = 1).

The objective is simply the sum of X for all time steps, because we discretized the time steps uniformly. We also add an imaginary query plan which requires all the indexes and makes the runtimes of all queries zero. This ensures the objective value is 0 for time steps that remain after all the queries are built.

This MIP model correctly solves the ordering problem but introduces many constraints and variables (it requires more than one million variables for large problems) due to non-linear properties of the problem. Unfortunately, MIP solvers cannot find feasible solutions to problems of this size after several hours of computation.

**APPENDIX III**

**DETAILED ALGORITHM FOR INITIAL SOLUTIONS**

Algorithm [1] provides the full greedy algorithm described in Section [VI-D]. We developed this algorithm to provide good initial solutions to the local search methods.

Algorithm [2] provides our implementation of the dynamic programming scheduling algorithm suggested in [15]. We used Stoer-Wagner Min-Cut [22] to divide the set of indexes into two sub-clusters. Our definition of edge weights between the
Let $G$ be the total cost of building the indexes of $L$ in the order $L$ specifies. As an abbreviation, we will use $C(i) \equiv C(i, \emptyset)$, e.g., $L_1 = i_1 \to i_2$ and $C(L_1) = C(i_1) + C(i_2, \{i_1\})$. Let $S(i, M)$ be the query speed-up of building $i$ assuming the indexes of $M$ are already built. We will also use the $S(i) \equiv S(i, \emptyset)$ abbreviation. Because the eventual speed-up achieved by the indexes does not depend on the order of indexes, the first parameter of $S$ can be a set of indexes unlike $C$.

Let $G_i$ be the basic area of index $i$. Trivially, $G_i = S(i, \ldots)C(i, \ldots)$. To simplify the notation, let us extend $G$ to subsequences as illustrated in Figure 14. Note that the second parameter of both $C$ and $S$ is the set of indexes built before. All indexes built after have no effect on the value of $C$ and $S$.

**Algorithm 2: DP: Dynamic Programming Algorithm** [15]

| Inputs : Index set $I$. Query set $Q$. |
| Outputs : Ordered list of indexes $N$. |
| if $|I| = 1$ then |
| return $I$; |
| end |
| // Cluster by Min-cut and Recurse |
| $I_1, I_2 = \text{MIN-CUT}(I)$; |
| $N_1 = DP(I_1), N_2 = DP(I_2)$; |
| // Merge sub-results by Interleaving |
| $N = []$; |
| while $N_1$ and $N_2$ are not empty do |
| benefit1 = benefit($Q, N \cup N_1$.front($));$ |
| benefit2 = benefit($Q, N \cup N_2$.front($));$ |
| if benefit1 > benefit2 then |
| $N$.append($N_1$.pop.front($));$ |
| else |
| $N$.append($N_2$.pop.front($));$ |
| end |
| $N$.append(remaining $N_1$ and $N_2$); |
| return $N$; |

Finally, let $R_M$ be the total query runtime when indexes in $M$ exist, namely $R_M = R_0 - S(M)$. For example, the total objective area (shaded area) in Figure 14 is

$$Obj(L_a \to i \to L_b) = G_a + R_{L_a}C(L_a) + G_i + R_{L_a+i}C(i, \{L_a\}) + G_b + R_{L_a+i+L_b}C(L_b, \{L_a+i\})$$

**The Swap Property:** Here we discuss a useful building block for the other proofs in this section. Consider the objective values of solutions $N = L_a \to L_i \to L_j \to L_b$ and $N' = L_a \to L_j \to L_i \to L_b$, which are identical except for the swap of $L_i$ and $L_j$. 

**Algorithm 1: Interaction Guided Greedy Algorithm**

| Inputs : Index set $I$. Query set $Q$. |
| Outputs : Ordered list of indexes $N$. |
| $N = []$; |
| while $I$ is not empty do |
| bestIndex = null; |
| foreach $i \in I$ do |
| benefit = 0; |
| foreach $q \in Q$ do |
| previous = $q$.getRuntime($N$); |
| next = $q$.getRuntime($N \cup i$); |
| benefit += previous - next; |
| // Add remaining interactions to benefit |
| foreach $p \in \text{plans}(q) : i \in p$ do |
| interaction = next - $q$.getRuntime($p$); |
| if interaction > 0 and $p \setminus N \neq \emptyset$ then |
| benefit += interaction / $|p \setminus N|$; |
| end |
| end |
| density = benefit / $i$.getBuildCost($N$); |
| if bestIndex = null or density > bestDensity then |
| bestIndex = density; |
| bestIndex = $i$; |
| end |
| $N$.append(bestIndex); |
| $I = I \setminus$ bestIndex; |
| end |
| return $N$; |
B. Alliances

**Definition:** Allied indexes are a set of indexes that only appear in query plans as a complete group and have no external interactions for building cost improvements.

**Theorem 1:** Every problem has at least one optimal solution in which allied indexes are built consecutively.

**Proof:** Let $i$ be the first created index among some allied indexes. Suppose a solution $N$ in which there is a non-empty subsequence $L_b$ between $i$ and its allied indexes, namely $N = L_a \rightarrow i \rightarrow L_j \rightarrow L_b$ where $L_b$ contains the allied index of $i$. Now, consider an altered solution $N' = L_a \rightarrow L_j \rightarrow i \rightarrow L_b$. We will prove the objective of $N'$ is always smaller or the same as that of $N$.

Because $i$ requires the allied indexes contained in $L_b$ to speed up any query, $G_i = G'_i = 0$ and $R_{L_a+i} = R_{L_a}, R_{L_a+i+j} = R_{L_a}$. By definition $i$ has no interactions that speed up building any index in $L_j$, therefore $G_j = G'_j$, and $C(L_j, \{L_a + L_i\}) = C(L_j, \{L_a\})$. Because $R_{L_a+j} \leq R_{L_a}, C(L_i, \{L_a + L_j\}) \geq C(L_i, \{L_a + L_j\})$, from (23),

$$\text{Obj}(N) - \text{Obj}(N') = R_{L_a}C(L_i, \{L_a\}) - R_{L_a}C(L_i, \{L_a + L_j\}) \geq R_{L_a}C(L_i, \{L_a\}) - C(L_i, \{L_a + L_j\}) \geq 0$$

Thus, a solution that does not create allied indexes consecutively can be improved by swapping so that the allied indexes come closer. By induction on the swapping of indexes an optimal solution can always contain a consecutive order of allied indexes.

**Detection:** We detect alliances in problem instances as follows. First, we list all interactions as candidate alliances. Second, for each alliance, we look for overlaps with the other candidates. In the example in Figure 4, $i_5$ overlaps between $\{i_1, i_3, i_5\}$ and $\{i_2, i_5\}$. If there is any overlap, we break the alliances into non-overlapping subsets. In the above case $\{i_1, i_3\}$, $\{i_2\}$ and $\{i_5\}$. We remove alliances with only one index, obtaining $\{i_1, i_3\}$ in the example. The detection overhead is $O(|P|^2)$.

C. Colonized Indexes

**Definition:** An index $i$ is called colonized by a colonizer index, $j$, iff all query plans using index $i$ also use the colonizer, $j$, and the index has no interaction to speed up building other indexes.

**Theorem 2:** Every problem has at least one optimal solution where every colonized index is built after its colonizer.

**Proof:** Let $i$ be a colonized index. Suppose a solution $N$ in which there is a subsequence $L_b$ between $i$ and its colonizer, $j$, namely $N = L_a \rightarrow i \rightarrow L_j \rightarrow L_b$ where $L_b$ contains $j$. Now, consider an altered solution $N' = L_a \rightarrow L_j \rightarrow i \rightarrow L_b$.

1 If there are not multiple optimal solutions (tie), each theorem simply means “every optimal solution should ….”
An optimal solution does not build $N' = L_a \rightarrow i \rightarrow j \rightarrow L_b$ which is no worse than all the other solutions that create indexes between $i$ and $j$.

Consider $N'' = L_a \rightarrow j \rightarrow i \rightarrow L_b$. By the same discussion, we show that $N''$ is no worse than $N''$ and may even be better.

Once again by induction on the swapping operation, any solution that builds a colonized index before its colonizer can be improved by moving the colonized index after its colonizer.

**Detection:** The detection algorithm for colonized indexes and its computational cost is quite similar to that of alliances. For each index, we consider all the query plans it appears in and take the intersection (overlap) of them, which is the colonizer(s). The detection overhead is $O(|I||P|)$.

### D. Dominated Indexes

In Section IV-C, we explained a simplified case of dominated indexes. Here we discuss dominated indexes in detail.

**Definition:** Index $i$ is dominated by index $k$ iff all of the following conditions hold. $\forall L_a, L_j, j \in L_j$ in $\{25\}$,

1. $S(k, \{L_a + L_j\}) \geq S(i, \{L_a + L_j\})$
2. $C(i, \{L_a + L_j + k\}) \geq C(k, \{L_a\})$
3. $C(j, \{L_a + i\}) \geq C(j, \{L_a + k\})$
4. $S(j, \{L_a + M + i\}) \leq S(j, \{L_a + M + k\}) : \forall M \in L_j, j \notin M$
5. $C(k, \{L_a + L_j\}) = C(k, \{L_a\})$

In short, $k$ is always more beneficial and cheaper to build than $i$. Note that these conditions are re-evaluated when some index is determined to be before or after $i$ or $k$ because indexes after both $i$ and $k$ are irrelevant to these conditions. At each iteration we re-evaluate these conditions to ensure maximum dominance detection.

**Theorem 3:** An optimal solution does not build $i$ before $k$.

**Proof:** Consider two solutions $N = L_a \rightarrow i \rightarrow L_j \rightarrow k \rightarrow L_b$ and $N' = L_a \rightarrow k \rightarrow L_j \rightarrow i \rightarrow L_b$. In this setting, $i$ and $k$ are single indexes. Therefore, in $\{25\}$,

$$G_i = C(i, \{L_a\})S(i, \{L_a\})$$
$$G'_i = C(i, \{L_a + k + L_j\})S(i, \{L_a + k + L_j\})$$
$$G_k = C(k, \{L_a + i + L_j\})S(k, \{L_a + i + L_j\})$$
$$G'_k = C(k, \{L_a\})S(k, \{L_a\})$$

Also by the definition of $S$ and $R$,

$$S(i, \{L_a\}) + R_{L_a + i} = R_{L_a}$$
$$S(i, \{L_a + k + L_j\}) + R_{L_a + L_j + L_j + k} = R_{L_a + L_j + L_j + k}$$
$$S(k, \{L_a + i + L_j\}) + R_{L_a + L_j + k} = R_{L_a + L_j + k}$$
$$S(k, \{L_a\}) + R_{L_a + L_b} = R_{L_a}$$

Applying these to $\{25\}$, we get

$$\text{Obj}(N) - \text{Obj}(N') = (G_j - G'_j) + R_{L_a}(C(i, \{L_a\}) - C(k, \{L_a\})) + R_{L_a}(S(i, \{L_a\}) - S(k, \{L_a\})) + R_{L_a}(L_j, \{L_a + i\}) - R_{L_a}(L_j, \{L_a + k\}) + R_{L_a}(L_j, \{L_a + i + L_j\}) - R_{L_a}(L_j, \{L_a + k + L_j\})$$

Because of the condition (3) and (4),

$$\ldots \geq R_{L_a}(C(i, \{L_a\}) - C(k, \{L_a\})) + R_{L_a}(i, \{L_a\}) - R_{L_a}(i, \{L_a + k\}) + R_{L_a}(i, \{L_a + i + L_j\}) - R_{L_a}(i, \{L_a + k + L_j\})$$

Because $C(L_j, \{L_a + k\}) \leq C(L_j, \{L_a\})$ and the condition

$$C(i, \{L_a + k + L_j\}) \leq C(i, \{L_a\})$$

(5) $C(k, \{L_a + i + L_j\}) = C(k, \{L_a\})$.

$$\ldots \geq C(L_j, \{L_a\}) - R_{L_a}(i, \{L_a\}) + R_{L_a}(i, \{L_a\}) - C(L_j, \{L_a\}) + R_{L_a}(i, \{L_a\}) - C(L_j, \{L_a\})$$

Now, by the definition of $S$,

$$S(L_j + k, \{L_a\}) = S(L_j, \{L_a\}) + S(k, \{L_a + L_j\})$$
$$S(L_j + i, \{L_a\}) = S(L_j, \{L_a\}) + S(i, \{L_a + L_j\})$$

From the condition (1), $S(k, \{L_a + L_j\}) \geq S(i, \{L_a + L_j\})$ and $S(L_j + k, \{L_a\}) \geq S(L_j + i, \{L_a\})$. Thus,

$$\ldots \geq S(L_j + i, \{L_a\}) - C(i, \{L_a + k + L_j\}) - C(k, \{L_a\})$$

From the condition (2), $\ldots \geq 0$.
**Detection:** We find dominant indexes in the following way. For each index, we calculate the minimum benefit and the maximum creation cost to make the indexes in each query plan. Then, we compare its ratio of minimum benefit to maximum cost with every other index’s ratio of maximum benefit to minimum cost. During this procedure, we consider the additional constraints to tighten the minimum/maximum. The detection overhead is $O(|T||P|)$.

**E. Disjoint Indexes and Clusters**

**Definition:** A disjoint index is an index that has no interactions with other indexes.

Let $den_i(M) \equiv \frac{S(i, M)}{C(i, M)}$ denote the density of $i$. Let, $L_a \rightarrow i \rightarrow L_b$ be an optimal solution. Suppose a suffix $L_j$ of $L_a$ such that $L_a = L'_a \rightarrow L_j$.

**Theorem 4:** Every suffix is more dense than $i$ if $i$ is disjoint.

**Proof:** We compare two solutions $N = L'_a \rightarrow i \rightarrow L_j \rightarrow L_b$ and $N' = L'_a \rightarrow L_j \rightarrow i \rightarrow L_b$. Because $i$ is a disjoint index, $G_i = G'_i, G_j = G'_j$, thus from (24),

$$Obj(N) - Obj(N') = C(i)(R_{L'_a+i} - R_{L_a+i}) - C(L_j, \{L'_a\})(R_{L_a} - R_{L_a+i})$$

$$= C(i)S(L_j, \{L'_a\}) - C(L_j, \{L'_a\})S(i) = S(L_j, \{L'_a\})\left(\frac{S(i)}{C(i)}\right) - S(L_j, \{L'_a\})\left(\frac{S(i)}{C(i)}\right)$$

The density of $i$ and $L_j$ is $den_i = \frac{S(i)}{C(i)}$ and $den_j = \frac{S(L_j, \{L'_a\})}{C(L_j, \{L'_a\})}$.

Hence, $Obj(N) - Obj(N') = S(i)S(L_j, \{L'_a\})(den_i^{-1} - den_j^{-1})$. Therefore, if $i$ has a larger density than any suffix, we can improve the solution by placing $i$ before the suffix which contradicts the optimality assumption of $N'$. 

**Theorem 5:** Every prefix is less dense than $i$ if $i$ is disjoint.

Let a dip be the place where we can place a disjoint index $i$ without violating the two theorems above. Now we prove that there is only one dip (except when there are tyes).

**Theorem 6:** Every sequence has only one dip to insert a disjoint index $i$.

**Proof:** Suppose there are two or more dips. Let $d_1 < d_2$ be the dips. Consider the sub-sequence $L_j$ between the places $d_1$ to $d_2$. From Theorem 4, $L_j$ has a larger density than $i$, but from Theorem 5, $L_j$ has a smaller density than $i$. By contradiction, there cannot be two more dips.

**Definition:** $i$ is backward-disjoint to $k$ iff all interacting indexes of $i$ and $k$ succeed $i$ or precede $k$.

**Definition:** $i$ is forward-disjoint to $k$ iff all interacting indexes of $i$ and $k$ precede $i$ or succeed $k$.

**Theorem 7:** An optimal solution does not build $k$ before $i$ if $i$ is backward-disjoint to $k$ and $den_i > den_k$.

**Proof:** Suppose $N = L_a \rightarrow k \rightarrow L_j \rightarrow i \rightarrow L_b$ is an optimal solution.

Consider the interactions $i$ and $k$ could have with $L_j$. Because $i$ is backward-disjoint, none of its interacting indexes are in $L_j$. Also, none of $k$'s interacting indexes are in $L_j$ either. In other words, $i$ and $k$ are disjoint indexes regarding the subsequence $L_j$.

Therefore, from Theorem 4 and Theorem 5, $k$ must be denser than $L_j$ and $L_j$ must be denser than $i$. However, by definition $den_i > den_k$ and we have a contradiction. Therefore, $N$ cannot be an optimal solution. As $L_a, L_j, L_b$ are arbitrary, and include empty sets, this means an optimal solution does not build $k$ before $i$.

**Theorem 8:** An optimal solution does not build $k$ before $i$ if $i$ is forward-disjoint to $k$ and $den_i < den_k$.

This proof is omitted as it is symmetric to the previous one.

**Detection:** We detect such cases as follows. For each pair of indexes, we check whether they are forward or backward disjoint to each other. If either of them is forward or backward disjoint, we can determine the interactions which $i$ and $k$ receive and calculate $den_i, den_k$. If the situation defined above occurs, we introduce the appropriate additional constraints. The overhead of this procedure is $O(|T|^2|P|)$.

**F. Tail Indexes**

**Definition:** Tail indexes are the last indexes to be built in a given build order $L$. Given some subset of indexes $M \in I$, we defined $M$’s tail group as all solutions where tail indexes are permutations of $M$. A tail champion of $M$ is the solution in $M$’s tail group that minimizes the tail’s objective.

**Theorem 9:** A tail champion of $M$ is better or same as all the other solutions in $M$’s tail group.

**Proof:** Consider the set of preceding indexes $A \equiv I \setminus M$ and its order $L_A$. Let us compare the objective of $N = L_A \rightarrow L_M$ and $N' = L_A \rightarrow L'M$. Suppose $N'$ is a tail champion of $M$’s tail group but $N$ is not.

$$Obj(N) = G_A + R_A C(L_A) + G_M + R_{A+M}C(L_M, A)$$

$$Obj(N') = G_A + R_A C(L_A) + G_M' + R_{A+M}C(L'M, A)$$

Now, because $N'$ and $N$ are in the same tail group and $N'$ is the tail champion, $G_M + R_{A+M}C(L_M, A) > G_M' + R_{A+M}C(L'M, A)$. Therefore, $Obj(N) - Obj(N') \geq 0$ for every possible $L_A$.

Let $F = \{M_1, M_2, \ldots\}$ be the set of all possible tail groups in the problem. Let $Const$ be a rule that holds in all tail champions of $M \in F$.

**Theorem 10:** $Const$ holds in the optimal solution.

**Proof:** From Theorem 9 the only possible optimal solu-
tion from $M$’s tail group is the tail champion. Because $F$ is a comprehensive set of all possible tail groups, the optimal solution is one of the tail champions.

Thus, regardless which tail group the optimal solution appears in, $Const$ holds in the optimal solution.

This theorem proves the property used in Section IV-E. We note that $Const$ can be any kind of rule. For example, “$i_1$ appears as the last index”, “$i_2$ is built after $i_3$”, “$i_3$ never appears in the last 3 indexes”.

**Detection**: At the end of each problem analysis iteration, we apply the tail analysis. We start from the tail length of 3 and increase the tail length until the number of tail candidates exceeds the threshold $k$. For each tail candidate, we calculate the tail objective and group them by the set of tail indexes as explained in Section IV-E. The detection overhead is obviously $O(k)$, thus $k$ is a tuning parameter balancing on the pruning power and the overhead of pre-analysis. In our experiments, we used $k = 50000$. 