QUERY-INDEX CO-OPTIMIZATION EXECUTING QUERY TEMPLATES FOR COMPLEX TEXT SEARCH

BY

RUI LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Kevin ChenChuan Chang

Abstract

Nowadays, many complex text search systems, such as Entity Search or Topic Search, have been proposed to allow users to retrieve fine granularity units (e.g., entities or topics) inside documents directly. As those search systems target on more complex search tasks, the traditional query processing method purely based on an inverted index can not execute those search queries efficiently. New execution algorithms and index structures need to be proposed.

In this paper, we study the problem of automatically deriving an efficient execution algorithm and indexes that support the algorithm for those systems. We take a relational view of the problem and model it as optimizing a query template with views. This query template optimization problem raises new challenges including *enumerating plans with views* and *selecting plans for answering a template* for a query optimizer. We present a novel optimization framework with a new set of transformation rules and an efficient selection strategy to deal with those two challenges.

We systematically evaluate our framework in two concrete application settings. Experiments show that: (1) The derived algorithm and indexes significantly improve the efficiency the keyword-based baseline method. (2) Our framework can automatically derive plans and indexes that are manually optimized for a system. (3) Our approach is general enough to be applied to different search systems.

To my family for all their love.

Acknowledgments

First, I would like to thank Professor Kevin Chang for his guidance and insightful comments for my research work. Second, I would like to thank all the faculties and colleagues in DAIS (Data and Information System) research group at the University of Illinois, Urbana-Champaign.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Complex Text Search System Overview

Nowadays ubiquitous text data, such as web pages or news archives, becomes an ultimate repository for people to find useful information. Everyday millions queries are submitted to web search engines to search relevant pages from the Web. In order to meet users'information needs directly, a new type of search systems [7, 17, 20], which allow to users to search specific information beyond web pages, have emerged. The following two systems are typical examples.

Scenario 1 ***EntitySearch System*****:** An entity search system (EntitySearch) enables users to search various types of entities (e.g., #university, #professor) directly from a document collection [7]. Each type of entities (e.g. "Stanford" $\in$ #unversity) are pre-extracted from each web page in a collection C. The system accepts a query containing both keywords K (e.g., "computer science") and an entity type E (e.g., #professor) and returns a list of entity instances ordered according to their relevance to the query. The relevance of an entity instance is calculated by how the entity matches with the keywords across all the pages.

Scenario 2 ***TopicSearch System*****:** A popular topic search system (TopicSearch) enables users to retrieve popular topics from a document collection (e.g. Twitter status). A topic can be any noun phrase or any term defined in a dictionary (e.g. Wikipedia). Topics are pre-extracted from each document and each document has a time stamp s_i (e.g., Sep 1, 2009). The system accepts a query containing both keywords K (e.g., "Obama") and a period p (e.g., Sep 01 2009), and returns a list of topics (e.g., "speech", "health care ") according to their popularity. The popularity of a topic is measured by how likely a topic associates with keywords across all the documents during the given period.

The above examples illustrate search tasks of two specific systems. We can find that although search tasks of different systems are different, they can be abstracted as the following three operations:

Fine Granularity Unit Retrieving Those systems retrieve a piece of useful text (e.g., entities and topics), which can be pre-extracted inside a document, instead of relevant documents. Since those pieces of text represent more specific information than documents, we define them as fine granularity units.

Contextual Matching When matching a fine granularity unit with a query, those systems need to examine the context of every document. A unit matches with a query only if the unit occurs with query keywords in a document. In addition, the matching of a unit and a query is evaluated across a collection since the unit may occur in many pages in the collection. For example, in a TopicSearch system, the score of a topic is defined based on how frequently it occurs with the keywords in the documents created at the given time period.

Complex Filtering Those systems allow users to specify additional filtering conditions for documents or fine granularity units. For example, in an EntitySearch system, an entity instance is matched if the instance is the given type and co-occurs with query keywords in a document. In a TopicSearch system, a topic is matched if the topic co-occurs with the given keywords in a document that is created at the given time period.

Therefore, the general function of those systems can be summarized as *retrieving fine Granularity units, such as entities and topics, inside documents directly and across many pages holistically with keywords and additional conditions.* Comparing to a traditional document search system, those systems have more complex search functions. We name them as *complex text search (CTS) systems*.

1.2 Computation Challenges

As an online search system, a CTS system should execute a query efficiently. However, those systems cannot efficiently be supported by the standard document search procedure, which retrieves a set of relevant document by intersecting inverted lists of each keyword. Additional computations are needed to retrieve fine granularity units from matched documents. Specifically, after retrieving all the relevant documents with keywords, a CTS system has to access the content of each relevant document to match fine granularity unites. This additional step accesses the content of each relevant documents and costs N random reads, where N is the number of relevant documents. It is inefficient when the N is large. Therefore, there is

a great demand for deriving an efficient online execution algorithm and corresponding new indexes that support the algorithm to efficiently support a CTS system.

In this paper, we propose to study the problem of automatically deriving an online execution algorithm and corresponding indexes for a CTS system. It is a nontrivial task due to the following challenges: First, for a CTS system, many kinds of indexes can be built, and many execution algorithms can be constructed with a given index configuration. Some of them are efficient while others may not. Thus, the solution should be able to systematically enumerate every candidate algorithm and indexes and select the most efficient algorithm and indexes used by it. Second, there are usually constraints on what kind of indexes can be used and how much space indexes can take. Thus, the solution should be able to handle those constraints when selecting online algorithms and indexes. Third, there are different CTS systems, each of which has a specific search task. The solution should be general enough to apply to different CTS systems.

1.3 Our Solution Overview

In order to derive an execution algorithm with indexes for a CTS system in a principled way, we take a relational view of the problem. Queries of a CTS system is viewed as a SQL template; the execution algorithm is viewed as a query plan for the template; and indexes are viewed as materialized views used in the plan. Thus, the original problem becomes a problem of *optimizing a SQL query template with views*. With this relational view, mature database query optimization framework can help us to systematically enumerate and select plans for a template, and different search tasks can be viewed as different templates, which can be optimized in a unified framework.

However, this new template optimization problem raises new challenges for a traditional query optimization algorithm. Specifically, the challenges are how to enumerate efficient plans with views systematically and how to select efficient plans for answering a template. We develop a general optimization approach to solve the new challenges. To enumerate plans with views, we identify ways of using views to construct improved plans for a given plan and model them as a set of new transformation rules. The new rules can be used to construct efficient plans with views in a systemical way. To answer all the queries represented by a template efficiently, a set of plans needs to be selected. Our approach selects the best plan set in a a cost estimation based approach. An efficient greedy selection strategy with guaranteed approximation rate is used in the selection procedure, so that the most efficient plan set can be constructed step by step.

Chapter 2

Problem Abstraction

2.1 A Relation View of CTS Systems

To design a general solution, which can derive an efficient execution algorithm and indexes for any CTS system, we take a relational database view of a CTS system. Specifically, we abstract search tasks of different CTS systems as standard database operations over a relational schema.

2.1.1 From a document collection to a database schema

As discussed in Section1, search tasks of different CTS systems can be abstracted as searching fine granularity *units* from *documents* with *keywords*. Documents, keywords, units and their relations can be captured by a relation model shown in Figure 1. Specifically, a document is an entity with attributes (e.g., time). It can be captured by a document table $Doc(docID, att_1, ..., att_n)$. Documents also contain keywords (e.g., Microsoft, kick). This relationship can be captured by a keyword and document relation table $KeyDoc(docID, keyword)$. Since a keyword does not have any additional attribute, we do not need to use a sperate table for keywords. Documents also contain units(e.g., Windows). This relationship can be captured by a unit and document relation table $UnitDoc(unit, docID)$. A unit is an entity with attributes(e.g., entity type). It can be captured by a unit table $Unit(unit, att_1, ..., att_m)$. Formally, we define the following a CTS schema to capture keywords, documents, units and their relations in different CTS systems.

Definition 1. CTS Schema *is a relational schema that captures keywords, documents, units and their relations of a CTS system. It contains four tables: a document table* $T_1 : Doc(docID, att_1, ..., att_n)$*, a keyword document relation table* $T_2 : KeyDoc(docID, keyword)$*, a unit document relation table* $T_3 : UnitDoc(unit, docID)$*, and a unit table* $T_4 : Unit(unit, att_1, ..., att_m)$.

Example 1. *The CTS schema for a TopicSearch system contains* T_1,T_2*, and* T_3 *defined in Definition 1. In*

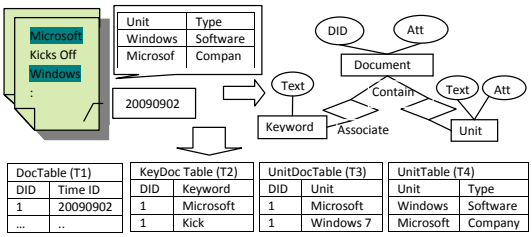


Figure 2.1: The Relation Tables

particular, T_1 describes the time stamp for each document, and T_3 describes which document a topic is extracted from. Since a topic does not have any attribute, the schema does not have a unit table T_4.

2.1.2 From search queries to a SQL template

Given the CTS schema, the task of searching fine granularity units from a document collection with a query can be viewed as executing a SQL query on those tables. Specifically, searching documents with a keyword can be viewed as a selection over *KeyDoc*. Searching documents with a set of keywords can be viewed as intersections over selected document list of each keyword. Contextual matching units with keywords in the context of a document and across many documents can be viewed as joining on *UnitDoc* and selected documents of the keywords. Filtering documents with a time period or filtering entity instances with a type can be viewed as a selection over the *Doc* and *Unit* respectively.

Furthermore, the computation for different queries from a CTS system are basically same except that the number of keywords and specific values of keywords and filter conditions are different. Queries with various numbers of keywords can be captured by a K-nary intersection operation, which takes a variable number of lists as input and computes the intersection of those lists. Different values of keywords and filter conditions can be represented by a set of parameters, each of which represents a set of possible values. Therefore, we can use a query template with variable-length parameters to represent all the queries.

Definition 2. CTS Query Template *is a SQL query defined based on a set of relational operations including selection (σ), projection (π), join ($\bowtie$), intersection ($\cap$) or union ($\cup$). Each selection condition is a parameter, denoted as $\sigma_{A=X}(T)$, where X is a parameter and A is an attribute of T And $\bowtie$, $\cap$, $\cup$ may be a K-nary operation, which takes a variable inputs.*

We use $Q(\{T_1.A_r = X_1...\}, ..., \{T_k.A_s = X_i...\})$ to denote a template with variable-length parameterized selections. Each $T.A = X$ means a parameterized selection $\sigma_{Att=X}(T)$ in the query, which can be instantiated

to any value in the attribute A. $\{T_1.A_{=}X_1...\}$ means the selection may repeat variable times. In this paper, we use capitalized X to represent a parameter, and the lower cased character x to a specific value of a parameter.

Given a template for a CTS system, any query is an instance of the template. Thus, the search task of a CTS system can be viewed as executing a SQL template.

Example 2. *In a TopicSearch system, retrieving topics for a query "Google, 2009" can be viewed as a SQL query, "select $T_1.DID$, TID from T_1, T_2, T_3 where Key = Google and $Time$ = 200909 and $T_1.DID = T_2.DID$ and $T_1.DID = T3.DID$". All the queries can be represented by a template $Q(\{T1.Key = k_1...\}, ..., \{T_2.Time = X\})$ shown below with parameters k_i presenting the kth keyword and p representing the time. In the template, intersection is a K-nary operation, which takes a variable number of lists as input.*

SELECT TID, DID FROM T_3, T_1
SELECT DID FROM T_2 WHERE Key = k_1 INTERSECT
...
SELECT DID FROM T_2 WHERE Key= k_n
WHERE DID = T_3.DID and TIME = p and DID = T_1.DID

2.2 A Relational View of the Problem

With a relational view of a CTS system, we further model the problem of deriving an efficient online execution algorithm with indexes for a CTS system as optimizing a query template with views.

2.2.1 From an online algorithm to execution plans

Since queries in CTS system is viewed as a SQL query template, an online execution algorithm for the query can be viewed as an execution plan for a SQL query. The problem of searching an efficient online algorithm for all the queries in a CTS system can be viewed as selecting the most efficient execution plan for the template.

In the database area, query optimization [10, 19, 4] is used to select the most efficient plan for a single SQL query. However, the most efficient execution plan for a template may not exist. Because the template represents a set of queries, the best plan for different queries is different. For instance, given a template, $(\sigma_{Key=k_i}T_2 \bigcap \sigma_{Key=k_j}T_2) \bowtie T_3$, the hash join method is used when the number of tu-

ples in $\sigma_{key=k_i}T_2 \bigcap \sigma_{Key=j}T_3$ is small; while the sort-merge join is used when the number of tuples in $\sigma_{key=k_i}T_2 \bigcap \sigma_{key=k_j}T_2$ is large. Therefore, to optimizer a template, a set of plans instead of a single plan should be selected. Each plan in the set can execute some queries represented by the template efficiently.

If a set of plans are selected for a template of a CTS system, an online execution algorithm for the CTS system can be easily built based on those plans. Specifically, the algorithm contains a plan selector and a set of routines, each of which is an implementation of a selected plan. When a query comes, the selector estimates the efficiency of each routine and uses the most efficient one to answer the query.

2.2.2 From indexes to materialized views:

In addition, the original problem needs to search an online algorithm with indexes. Indexes used by an online execution algorithm can be viewed as data structures for speeding up an execution plan. In the database area, both indexes and materialized views are such structures. Indexes, such as B-tree and multidimensional indexes, are pre-computed data structures that map a key to corresponding tuples. They enable efficiently accessing a subset of tuples with a key. Selection and join operations can be efficiently computed with indexes. Similarly, a materialized view pre-computes results for some operations and stores the results as a concrete table on a disk. Queries containing those operations can be answered efficiently via accessing the view directly.

Although indexes and materialized views are different concepts, an index, which maps a key to a subset of tuples associated with that key, can be conceptually considered as a set of materialized views [1], each of which pre-computes and stores results for a selection with a possible key on a single table. A "view" defined on a SQL template, also represents a set of materialized views, each of which pre-computes and stores results for a query instance of the template.

In this paper, we use *parameterized view* to represent both index and a view for a template.

Definition 3. Parameterized View *is a virtual view,* $View(\{T_1.A_r = X_1...\}..., \{T_k.A_s = X_i...\})$*, with parameters. It is defend by a SQL query template* $Q(\{T_1.A_r = X_1...\}..., \{T_k.A_s = X_i...\})$

A parameterized view represents a set of view instances. Each of view instance is corresponding to a view which materialize a query instance of the template with each parameter instantiated to a specific value in the corresponding attribute.

To represent an index over a table T using a parameterized view, the view materializes a selection operation on the table, where the selection condition is a parameter and the parameter can be instantiated to any possible key value. Specifically, it can be denoted as $\sigma Key = KT$.

Example 3. *A parameterized view $View(K) = \sigma_{Key=K}(T_2)$ with a parameter K represents a set of view instances, each of which materializes the selection operation with K instantiated to a possible keyword. Conceptually, each view instance stores documents associated with a particular keyword. This parameterized view can be viewed as an inverted index for a document collection.*

Based on the definition, a view that materializes operations is also a parameterized view, where parameters can be instantiated to any possible value of attribute used in those operations.

Thus, to search an execution algorithms with indexes, can be viewed as search plans that use parameterized views. Views used in the selected plans can be materialized as indexes for a CTS system. In the rest of paper, we use parameterized views and views interchangeably.

2.2.3 Problem Constraints

Based on the above discussion, the original problem becomes selecting a set of efficient execution plans with and without views for a query template. However, not all the views can be used to construct plans, and not all the plans can be selected. Otherwise, the most efficient way to execute a template is materializing a view for each query and using it to answer the corresponding query. It is unpractical because it costs too much space to materialize results for every query. Considering the setting of the original problem, we abstract two constraints in the new problem.

The first constraint, denoted as C^V, specifies that a view $View(\{T_1.A_r = X_1...\}..., \{T_k.A_s = X_i...\})$ can not have variable parameters. many view instances. Specifically, $|D(p_1, ..., p_n)| < N$, where D is domain of parameters and N is a predefined threshold. When the domain of parameters of $View(p_1, ..., p_n)$ is large, a lot of instance views need to be materialized. It takes a lot of space to materialize so many views and it is inefficient to locate a specific instance view from them. In the original problem, the query space consists of different keyword combinations. It is impossible to materialize views for all different keyword combinations, so the first constraint limits the number instances that needed to be materialized for a parameterized view. It is similar to a document search system, where an inverted list is built for each single term.

The second constraint, denoted as C^P, specifies that a selected plan uses at most B space and at most K plans can be selected. A plan with views requires additional space cost for storing the views. The disk space is not an unlimited resource in any system. Thus, a plan cannot take too much space. Furthermore, we can not select a lot of plans. If many plans are selected, the online selector will take a lot of time to search the most efficient one. It also takes additional space to store views used in them.

Based on the above discussion, we can formally define our problem in the following way.

Definition 4. Query Template Optimization with Materialized Views Problem: *The input of the problem is a template Q on a relational schema for a CTS system. Let $V = \{v_1, v_2, ..., v_n\}$ be all the materialized views that satisfy view constraints C^V, our problem is to search $P = \{p_1, p_2, ..., p_m\}$, which contains all the plans with and without views in V for executing Q, and to output a subset EP of P, such that the efficiency of answering all the queries of the given template, denoted as $E(EP, T)$, is the maximized among all the subsets $EP_i \subset P$ that satisfy the constraint C^P.*

With the above problem definition, we formally transfer the problem of deriving an online algorithm and indexes as a problem of optimizing a SQL template with views under the constraints. With this view of the problem, we can adopt the traditional query optimization idea to solve the problem efficiently.

2.3 Problem Setting Review

To help reader to understand our problem setting, Figure 2.2 gives an overview of our approach. . Via taking a relation view, we view data model and queries of a CTS system as a relation schema and a SQL template respectively. Then the problem of searching efficient online algorithm becomes the problem of optimizing a SQL template with views offline. The output of the query template optimization problem is a set of plans with and without views.

The selected plans are implemented as online execution routines. The views used by the selected plans are materialized as indexes. The execution engine uses a plan selector and those routines to answer queries. When a query comes, the engine uses the selector to select the most efficient routine, and then uses the selected routine to execute the query.

There are several advantages for modeling the original problem as optimizing an query template with views. First, the approach is general. Different CTS systems are captured by different SQL templates. It

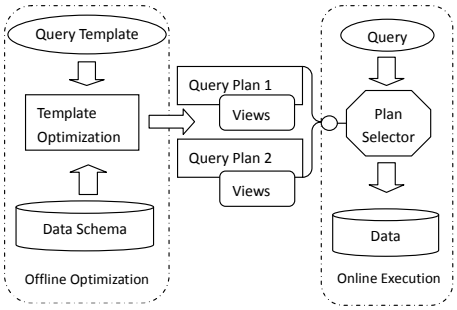


Figure 2.2: The Overview of Our Problem Setting

may be extended to other complex text search systems. Second, query optimization have been well studied in the database literature. Mature database technologies may be adopted to solve a solution.

Furthermore, we emphasize that our template optimization approach works *offline*. It is different from the traditional *online* query optimization, where an optimizer enumerates different plans and selects the most efficient plan for a query during the online execution stage. Comparing with online query optimization, our offline template optimization approach has the following reasons: (1) It can enumerate candidate plans extensively without worrying about the limited online optimization time. (2) It can enumerate plans with views, which can further be materialized to support online execution. (3) It directly prunes a large set of unselected plans for the online algorithm. The online algorithm can select the most efficient plan efficiently.

Chapter 3

Query Template Optimization

3.1 Solution Overview

In the database area, different query optimization [10, 19, 4] algorithms have been proposed to derive the most efficient execution plan for a SQL query. Generally, a query optimizer systemically enumerates a set of plans, which are formed by examining different execution orders, possible access methods and various execution methods. Then it attempts to select the most efficient one according to an estimated "cost" for each plan. However, those algorithms can not directly be applied to our problem because of the following challenges:

The first challenge is that an optimizer for our problem needs to enumerate execution plans with all possible materialized views. The traditional query optimizer can enumerate plans with available indexes, but it cannot enumerate plans with views. Although some optimization algorithms [4] can search efficient plans with views, they assume that views are given. Thus they only search plans with the given views. In our problem, no view is given beforehand. The optimizer needs to search plans with all the materialized views in additional to plans without views. Given a schema, numerous views can be defined; and even more plans can be derived with those views for a query. The search space of candidate plans is much larger than the search space of a traditional optimizer. For example, given a query $\sigma_{Key=k_i} T_2 \bowtie T_3$, $V_1 : T_2 \bowtie T_3$, $V_2 : \sigma_{Key=k_i}(T_2)$, and $V_3 : \sigma_{Key=k_i}(T_2) \bowtie T_3$, are some of possible views. $P_1 : \sigma_{Key=k_i}(V_1)$, $P_2 : (V_2 \bowtie T_3)$, $P_3 : \sigma_{Key=k_i} V_2 \bowtie T_3$, $P_4 : \sigma_{Key=k_i}(T_2) \bowtie V_2$, and $P_5 : V_3$ are some of candidate plans. Obliviously, we can not enumerate all the materialized views and all the plans with views. Our optimizer needs to find a way to systematically search only efficient plans with materialized views.

The second challenge is that an optimizer for our problem needs to select the most efficient plan set under the constraint for a given template, while a traditional optimizer can only select the most efficient plan for a single query. In our problem, the optimizer needs to define and estimate the efficiency for a set

plans for a template first; then selects the most efficient set in an efficient way. If the optimizer enumerates every possible set and selects the most efficient one, it will not be efficient because there are exponential number of candidates to be examined. For example, given N candidate plans, there are N^K candidate sets to be examined. Thus, our optimizer needs to find a way to efficiently select K plans that answer the template most efficiently.

Therefore, we need to design a new optimization framework, which can deal with the above two challenges. Although the traditional optimization algorithms can not solve our problem directly, we can adopt their framework to our problem. Specifically, our optimization algorithm works in two stages as a traditional query optimizer. First, it enumerates all the candidate plans with and without views. In this step, we identify a set of new transformation rules, which can derive efficient plans with views systematically. Second, It selects the best plan set from candidate plans in a cost estimation based approach. In this step, we implement an efficient selection strategy, so that the most efficient plan set can be constructed step by step. We will discuss both modules in detail. Through this section, we use the PTS schema and a query $Q : (\sigma_{Key=k_i}(T_2)) \bowtie T_3$ as a toy example to illustrate our idea.

3.2 Preliminaries

We develop our query optimization algorithm based on an existing query optimization framework, called Volcano [10], because new transformation rules can be easily plugged into the Volcano optimization framework. The Volcano uses a compact data structure to represent different candidate plans and uses a set of transformation rules to enumerate different plans. We briefly descries the compact structure and general steps of a Volcano based optimizer.

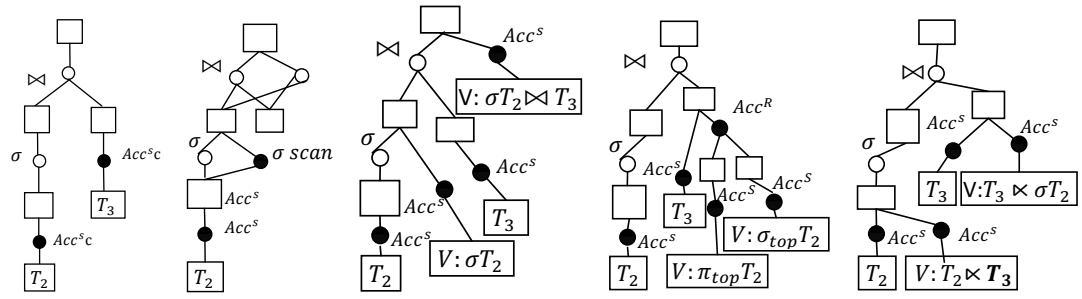
An AND-OR DAG is a compact data structure to represent different plans. An AND-OR DAG is a directed acyclic graph whose nodes can be divided into AND-nodes and OR-nodes. An *AND-node* presents an algebra operation such as join or selection, or an access method. There are two types of AND-nodes: (1) a logical AND-node, that maps to a relational algebraic operation (e.g., join), and (2) a physical AND-node, that represents a particular execution method of an algebra operation (e.g., hash join or sorted-merge join) or an access method (e.g., sequential access). An *OR-node* is either a table, called *table OR-node*, or a parent of a set of logical equivalent AND-nodes, called *intermediate OR-node*. The intermediate OR-node represents a sub-goal of a query plan. Each AND-node represents an execution plan for the OR-Node.

Based on the AND-DAG structure, an Volcano based optimizer works in the following way: First, the optimizer parses the query into a logical tree, which is an AND-OR DAG form. For example, the initial DAG for $\sigma_{Key='a'}(T_3) \bowtie T_2$, is shown in Figure 3.1(a). OR-nodes are shown as boxes, and AND-nodes are shown as circles. An empty circle represents a logical AND-node, while a black circle represents a physical AND-node. Acc^S means a sequential access method for a table.

Once the initial DAG is constructed, the optimizer applies transformation rules to derive new alternative nodes for each node in a bottom up fashion. A *transformation rule* transfers a DAG structure rooted at an AND-node into an equivalent DGA structure rooted at another AND-node. The new AND-node is a sibling of the original AND-node under the same OR-node. It represents a new execution plan. For example, the commutative law for join can be represented as $ORN_l \bowtie ORN_r \rightarrow ORN_r \bowtie ORN_l$, where ORN represents an OR-Node, and the right arrow means creating a new AND-node as an sibling of the original AND-Node. The physical plan also be generated via transformation rules. For instance, $\sigma_C(ORN) \rightarrow \sigma_C^{scan}(ORN)$ specifies that a new physical plan with the scan-based selection can be created based on a logical selection. Figure 3.1(b) shows a partially expanded DAG structure for the DAG in 3.1(a) with the above rules. It contains additional plans, such as $T_2 \bowtie \sigma_{Key='a'}(T_3)$ or $T_2 \bowtie \sigma^{scan}_{Key='a'}(T_3)$, for the query.

When the optimizer applies transfer rules to generate a new plan, it checks whether a rule can be applied to a node or not , and whether the new plan has been generated or not. After applying rules to the DAG, the expanded DAG structure represents all the different execution plans.

Then, the optimizer estimates costs of each node in a bottom up fashion, and prunes inefficient AND-nodes in each OR-node. The optimal query plan can be directly selected via recursively choosing the most efficient plan from its children. The detail of the approach can be found in [10].



(a) DAG (b) DAG (c) Rule 1 and 2 (d) Rule 3 (e) Rule 4

Figure 3.1: DAG Structures

3.3 Generating Plans with Materialized Views

We first discuss the enumeration module, which enumerates all the possible plans without and with views for a template.

As discussed before, the search space of all the possible plans without and with views is much larger than the search space of plans without views. To avoid searching such a large space and maintain the correctness of our solution, our enumeration algorithm only enumerates plans without views and plans with views that can improve efficiency of those without views. Plans that are less efficient than the plans without materialized views, or plans that are less efficient than the plans with the same set of views will be directly pruned from the enumeration space.

Based on the above idea, our optimizer can enumerate candidate plans without and with views in the following way. First, the optimizer enumerates all the plans without materialized views. Then, for each derived plan, the optimizer recursively derives all the plans that can improve the plan with additional views. Since plans without views can be enumerated by a traditional query optimizer, we will focus on enumerating plans that can improve a plan with different views.

In order to identify different ways of improving a plan with views, we examine factors that affect efficiency of a plan. They are the execution order of operations in a plan (e.g., $\sigma_{Key=k_i}(T_2) \bowtie T_3$ or $\sigma_{Key=k_i}(T_2 \bowtie T_3)$, execution methods of each operation (e.g., hash join or sorted-merge join), and accessing methods of each table (e.g., sequential scan or random index scan). A plan can be possibly improved if any of those factors is changed. Given a plan, its execution order is determined, and can not changed by using a view. Thus, ways of improving a plan with views are: (1) providing additional execution methods with views for any operation, (2) providing additional accessing methods with views for any table in the plan. Considering a simple case, where a plan contains a single operation and a table, a plan can be improved with views in the following three ways.

1. The first way is using a materialized view to replace the operation. Thus, an additional execution method for the operation is obtained. The new method materializes the operation directly with a view. The cost of computing the operation could be saved via accessing the view directly.

2. The second way is using materialized views to access the table randomly instead of scanning the table sequentially. Cost of scanning a table sequentially could be saved by randomly accessing subsets of

a table. Specifically, the plan can access a set of materialized views, each of which is a subset of a table. It can be logically considered as randomly accessing the table with an index.

3. The third way is sequentially accessing a view instead of the original table. The plan can be improved if the view is smaller than the table and can produce the same results as the original table.

The above three ways then can be generalized as transformation rules to generate improved plans with views for each plan. With those additional rules, we extend the Volcano based optimizer to enumerate efficient plans without and with views. Next, we discuss those rules in detail.

3.3.1 Materializing operation with a view

The first way to improve an execution plan with a view is materializing an operation with a view. Materializing an operation as a view provides an additional execution method. The new plan can directly access the view to obtain results of the operation. The cost for computing an operation online is saved by materializing the operation offline. In this way, the view used in the plan is the view that materializes the operation. We formalize the above idea as the following transformation rules.

$$Rule1 : (Acc^S(T_l))OP_a(Acc^S(T_r)) \rightarrow Acc^S(View((T_1)OP_a(T_r))) \tag{3.1}$$

$$Rule2 : OP_u(Acc^s(T)) \rightarrow Acc^S(View(OP_u(T))) \tag{3.2}$$

where OP_a represents an associated operator such as join or intersection; and OP_u represents a unary operator such as selection or projection. T_l, T_r, or T represents a table OR-node, which could be a table or a view. Acc^S means a sequential access method for a table. $View$ means materializing as a view.

When our optimizer applies these rules, there are two constraints. The first one is that an operation can not be materialized if any of its children OR-nodes does not contain the AND-node that is accessing a table. It is because we can not materialize an operation, if any of its operands depends on results from some online operations. The second constraint is that an intersection operation can not be materialized due to the constraint C^V defined in our problem.

Example 4. *Figure 3.1(c) shows the expanded DAG after applying the above rules to the toy example.*

Specifically, the second rule is applied to the selection. A new physical AND-node, which sequentially accesses $View(\sigma_{A='a'}(T_1))$, is generated as a sibling of the original selection node. The first rule is applied to the join. A new AND-node, which sequentially accesses $View(\sigma_{Key=k_i}(T_2) \bowtie T_3)$, is generated as a sibling of the join node. The DAG now contains additional plans with views.

3.3.2 Random accessing with views

The second way to improve a plan with views is to provide a random access method for a table. Traditionally, indexes are used to enable random accessing a subset of a table with keys. The cost of a plan is saved by random accessing a subset of a table directly instead of sequential accessing the entire table.

Inspired by the idea of an index, we can use views to function as an index and to enable random accessing a table. Specifically, any index can be viewed as a structure that maps a key to a subset of tuples associated with that key. Given a table, a set of tuples associated with a key can be defined as a view containing a selection operation on the table where a key is the selection condition. Then a set of views, each of which materializes results for a possible key value, can be conceptually viewed as an index. Given a set of keys, a plan will sequentially access views associated with each key. It can be conceptually viewed as random accessing the original table.

To represent the random access method with views, we use $\sigma_{Key}(T)$ to represent views associated with key values, and use $\Pi_{Key}(T)$ to denote all the values for the key attribute. Thus, the random accessing method for a table T with some attributes as the key can be represented as

$$Acc^R(Acc^S(View(\Pi_{key}(T)), Acc^S(View(\sigma Key = k_i(T)); \tag{3.3}$$

Acc^R presents the random access method. To random access the entire table T, a plan needs to fetch every possible key value for the table by sequentially accessing $View(\pi_{key}(T))$. For each key value, the plan accesses tuples associated with it by sequentially accessing $View(\sigma Key(T)$. Furthermore, there are multiple choices for keys. Any subset of attributes of a table could be a candidate. For example, either $Topic$ or Doc can be used as the key to randomly access T_3. Thus, there are different random access methods for a table. Each method uses a different key.

According to the above discussion, with a set of views, additional random methods can be obtained. We

formalize this idea as the following rule.

$$Rule3 : Acc^S(T) \rightarrow Acc^R(Acc^S(View(\Pi_{key_i \subset att(T)}(T))), Acc^S(View(\sigma_{key_i \subset Att(T)}(T)))); \quad (3.4)$$

where $Att(T)$ represents attributes of T. The rule creates a set of new And-nodes. Each node represents a new random access method for the original table T with key_i.

Example 5. *Figure 3.1(d) shows the DAG after applying the rule to the toy example. It generates a new access method for T_3, which randomly accesses T_3 using $Topic$ as the key. Two views are used. $\Pi_{Topic}(T_3)$ is to get each possible value of the key; and $\Sigma_{Topic}(T_3)$ is to get tuples with each key.*

In addition, since those views play a role as an index. Our optimizer will further enumerate plans with indexes based execution methods (e.g., index selection, index join) when those views are available.

For this rule, we emphasize some points here. First, this rule is different from the rule that generates the index-based execution method. They are different because they generate different plans. For example, $(\sigma_{key=k_i}(T_2) \bowtie Acc^R(Acc^S(View(\Pi_{Topic}(T_3))), Acc^S(View(\sigma Topic(T_3)))))$, which access tuples in the T_3 with each topic, is not any index based hash join method. Although this plan seems inefficient at first, the plan can be further improved with combining other rules. Second, we assume the tuples associated with a key is sequentially stored in a materialized views, because this will minimize the cost of accessing the data in a view. This will be equal to the case where index files are clustered. And with this representation, we do not model how to point the view for a key. This depends on the specific index structure used for storing the pointers, such as B-tree or hash-index. We will assume the optimal case, where the pointer can be obtained with a random read. This rule can be easily extended to model different index structures.

3.3.3 Sequential Accessing with Filter Views

Intuitively, some tuples in a table may not satisfy conditions of some operations in a query, those tuples will not be used to compute the final results. A view filters out those tuples from the table can still generate correct tuples for the query. For example, a view $View(T_2 \ltimes T_3)$ can replace T_2 to compute the query correctly, since all the tuples that can join with T_3 are kept in V. Thus, a plan can be improved if the new plan accesses this kind of views instead of the original table. Since the view is smaller than the table, the cost of the plan will be reduced.

In this way, a view plays a role as a filter for a table. We name it as a filter view. Formally, we can define a view V is a *filer view* of a table T for a query Q as a view satisfies $V \subset T$ and $Res(Q(V|T) = Res(Q(T))$; where $Q(V|T)$ denotes a new query, which replaces T in Q with V. $Res(Q)$ denotes the result of Q.

Based on the definition, there are many filter views of a table for a given query. We use $T \ltimes Res(Q)$ to represent all the tuples in T that are kept in the result of Q. Any view V that satisfies $V \supset T \ltimes Res(Q)$ and $V \subset T$, is a filter view. For example, views, such as $View_1(T_2 \ltimes T_3)$, $View_2(T_2 \ltimes \sigma_{Key='k_i'}(T_2))$ and $View_3(T_2 \ltimes \sigma_{Key='k_i'|Key='k_j'}(T_2))$, are all filter views for T_2.

However, some of candidate filter views are not easy to derive. It requires additional logical inferences to determine whether a view is a superset of $T \ltimes Res(Q)$. For example, determining $View_3$ is a filter view for T_2 requires additional logical inferences. But those views may not be as useful as views that can be directly derived from operations in the query. $View_3$ is not as useful as V_2 since V_2 contains less tuples than V_3. In this paper, we use a closed world assumption. Specifically, we only consider filter views that can be defined by operations in the query. In the previous example, $View_1$, $View_2$ are considered as filter views of T_2. Under this assumption, filter views of a table T are defined as

$$FilterView(T, Q) = \{V|V = T \ltimes Res(Q'); \forall Q' \subset Q\}; \tag{3.5}$$

where Q' is a sub query of Q. A filter view V of a table T is a view that contains tuples that at least match some operations of the given query.

Based on the definition, filter views for a table can be generated accordingly. Specifically, given a query, the algorithm uses a traditional query optimizer to enumerate different plans of a query. The sub queries can be directly obtained as subtrees as those plans. Filters views of a table can be derived by semi-joining the table and each sub query. Duplicate or redundant views are removed in the end. In the toy example, sub queries of the query are: T_2, T_3 $\sigma_{Key='k_i'}(T_2)$ $(T_2) \bowtie T_3$, $\sigma_{A='k_i'}(T_2) \bowtie (T_3)$, the corresponding filter views for T_3 after removing duplicates are: $Viiew_1(T_3 \ltimes T_2)$, $View_2(T_3 \ltimes \sigma_{Key='k_i'}(T_2))$. If a view contains the same set of tuples as the table, the view is pruned.

Given a set of filter views for a given table, the original plan can be improved by replacing the table with one its filter view. Specifically, a set of new plans can be generated with the following rules.

$$Rule4 : Acc^S(T) \rightarrow Acc^S(V); \forall V \in FilterViews(T, Q). \tag{3.6}$$

where $FilterViews(T, Q)$ are the set of filter views of T for Q. The rule creates a set of new AND-nodes as additional sequential accessing methods for the table.

Due to the constraint of our problem, we can not materialize any intersection in a view. A filter view can not contain an intersection either. However, intersection may help to filter out a lot of tuples. In order to preserve the filter idea, we add a rule to deal with a filter view containing an intersection operation. Specifically, the optimizer transfers the filter view into an AND-Node which intersects a set of filter views without the intersection operation.

Example 6. *Figure 3.1(e) shows the DAG after applying the rule. Specifically, the rule is first applied to* $Acc^S(T_2)$ *and generates a new access method with a filter view* $View(T_2 \ltimes T_3)$*. The rule is then applied to* $Acc^S(T_3)$ *and generates* $Acc^S(View(T_3 \ltimes \sigma(T_2))$*.*

3.3.4 Summary

With the above transformation rules, we extend the Volcano based optimizer to enumerate improved plans with views for a plan. The following theorem shows the completeness of those rules in searching improved plans with materialized views for a plan when the closed world assumption is not used.

Theorem 3.3.1. *Given a plan P with a determined execution, if the plan only contains* $\ltimes$*,* Σ*,* $\cap$*, and* π*, the rules are complete in generating improved plans that have the same execution order with materialized views.*

To prove the theorem, we use a tree structure to represent a plan. Based on the tree structure, we prove this theorem with the structure induction technique. The detail of the proof can be found in 7.1.

The theorem states that the rules are complete in generating improved plans with materialized views for a plan with the determined execution order. To generate all the efficient plans with views, our optimizer can generate all the plans that have different execution orders by a traditional query optimizer without considering views. Then the additional rules are applied to generate all the efficient plans with materialized views for the template. The detail of implementation of the optimizer is discussed in Appendix 3.5.

We briefly emphasis several points here: (1) Some other additional rules are plugged into the traditional query optimizer, so that it can fully search the plans with different execution orders. (2) All the rules are recursively applied to the DAG in order to generate all the candidate plans. A newly generated plan can be

further improved with views. (3) Some rules may generate the same plan. The duplication can be handled in two ways. First, we always check whether the plan has been generated or not. Second, several heuristics can be applied to avoid generating duplicates.

3.4 Selecting Plans with Constraint

After enumerating all the possible plans, the optimizer needs to select the most efficient plan set from them to answer the template. Although selecting a set of plans is different from selecting a single plan in the traditional query optimization setting, we can still adopt a cost estimation based approach, which selects the most efficient candidate according to its estimated efficiency. In this subsection, we discuss how to estimate the efficiency of a set of plans for the template, and how to efficiently select the most efficient set according to the estimated efficiency efficiently.

3.4.1 Cost Estimation

We need to first define the efficiency of a set of plans for a template. Intuitively, it can be defined as average efficiency of a set of plans for all the query instances of the template. Let $E(S, Q)$ be the efficiency of a set of plans S for the template Q. $E(S, Q) = \frac{\sum_{q_j \in Q} E(S, q_j)}{|Q|}$; where the template Q represents a set of query instances; q_j is a query instance; and $E(S, q_j)$ is the efficiency of using S to answer q_j. Intuitively, when a set of plans is used to answer a query, the most efficient plan from the set is selected. Therefore, $E(S, Q)$ is defined as the following equation.

$$E(S, Q) = \frac{\sum_{q_j \in Q}(\max_{p_i \in S} e(p_i, q_j))}{|Q|}; \tag{3.7}$$

where p_i is a plan in S; and $e(p_i, q_j)$ is the efficiency of using p_i to answer q_j. $e(p_i, q_j)$ can be measured as the reduced estimated cost, if p_i is selected to answer q_j instead of a baseline method (e.g., the inverted index based execution method described in Section 1). Specifically, $e(p_i, q_j) = ec(base, q_j) - ec(p_i, q_j)$. $ec(p_i, q_j)$ is the cost of using p_i to answer q_j. It can be estimated by a cost model used in a traditional optimization approach.

According to the above definition, the efficiency of a set of plans for a template is measured based on all the query instances of the template. However, it is impractical to enumerate all the queries for a template,

because a template may represent millions of queries. In order to measure $E(S, Q)$ efficiently, we use a sampling idea to estimate $E(S, Q)$ based on a set of query samples. The estimated efficiency of a set of plans S for a template Q is defined as:

$$\tilde{E}(S, Q) = \frac{\sum_{q_j \in \tilde{Q} \subset Q} (\max_{p_i \in S} e(p_i, q_j))}{|\tilde{Q}|}; \tag{3.8}$$

where $\tilde{Q}$ is a set of samples of query instances represented by a template Q. It can easily be obtained from query logs of a system.

We also need to estimate space costs of a plan, denoted as $Cost(p_i)$. Specifically, it can be measured as the total space that are taken by views used in a plan P_i. The size of a parameterized view is the sum of the space of each view instance of the parameterized view.

$$Size(View(A_1 = X_1, ..., A_i = X_i) = \Sigma_{x_1,...,x_n \in D(A_1,...,A_i)} Size(View(A_1 = x_1, ..., A_n = x_n)); \tag{3.9}$$

where $Size(View(A_1 = x_1, ..., A_n = x_n))$ is the size of a view, which materializes a SQL query. The size of the result of a SQL query can be estimated by an traditional cost model or be computed by an execution engine.

3.4.2 Greedy Plan Selection

According to estimated efficiency and space costs of a set of plans, the optimizer outputs the most efficient plan set under the cost constraint. It can be defined as the following optimization problem.

Definition 5. Best Plan Set Selection Problem *is to select a subset of plans S from a set of candidate plans P for the template Q such that the efficiency of S is maximized subject to constraints that S contains at most K plans and each plan's space cost $Cost(p_i)$ is less than a budget space B. Specifically,*

$$\begin{aligned} &maximize && E(S, Q); \\ &subject\ to && |S| \leq K,\ \forall S \subset P \\ & && Cost(p_i) \leq B,\ \forall p_i \in S \end{aligned}$$

In the above problem, the second constraint is easy to satisfy. The optimizer can easily prune all the plans

whose costs are larger than B first. The first constraint makes the problem be a combinatorial optimization problem, which can not be efficiently solved. The following theorem suggests the problem is a NP-hard problem.

Theorem 3.4.1. *Best Plan Set Selection Problem is a NP-hard problem.*

We can prove it by reducing the K set cover problem to our problem. Detail of the proof is in Appendix 7.2.

According the above theorem, there is no polynomial algorithm that can find the optimal solution for the problem. An exponential algorithm for finding the optimal solution works as follows. First, it prunes plans that are larger than the budget B from all the candidate plans. Then, it enumerates all the subsets that contain at most K plans; and evaluates their efficiency. Finally, it outputs the most efficient set. The complexity of the algorithm is $O(N^K)$, where N is number of candidate plans. Since there are many candidate plans, the algorithm is inefficient.

In order to find the best plan set efficiently, we use a greedy selection strategy shown in Algorithm 3.4.2 in our selection module. The greedy algorithm starts with an empty plan set $S_0 = \emptyset$, and iteratively adds the plan p_i, which maximizes the marginal efficiency, defined as $E(S_{k_1} \cup \{p_k\}, Q) - E(S_{k_1}, Q)$ at the ith step, and stops once the constraint does not be satisfied or the efficiency function can not be improved. The running time of Algorithm 3.4.2 is $O(K|P|)$, where $|P|$ is the number of candidate plans. It is much more efficient than the previous exponential algorithm.

Algorithm 1 Greedy Selection Strategy

Input: A Set of Execution Plan P, $\forall p_i \in P, cost(p_i) < B$
Output: A Set of Execution Plans S
$S = \emptyset$
for $i = 1; i <= k; k++$ **do**
  $p_k = \max_{p_k \in P - S_{k-1}} E(S_{k_1} \cup \{p_k\}, Q) - E(S_{k_1}, Q)$
  if $\max_{p_k \in P - S_{k-1}} E(S_{k_1} \cup \{p_k\}, Q) - E(S_{k_1}, Q) == 0$ **then**
    break;
  $S = S + P_k$
return S

Intuitively the selected plans based on the above strategy will perform well because the optimizer always selects the most efficient plan at each step. We can actually prove that this greedy strategy achieves a guaranteed approximation rate, $(1 - 1/e)$ for the optimal solution.

In order to prove that the greedy strategy solves our problem with a guaranteed approximation rate.

We model our problem as maximizing a submodular function under a cardinality constraint. In literature, the maximization of a monotone submodular function under a cardinality constraint problem can be solved near-optimally in polynomial time [15].

In order to prove that the best plan selection problem is a instance of the maximization of a monotone submodular function under a cardinality constraint problem, we have to prove that the target function $E(S, Q)$ in our problem satisfies the following properties. Firstly, the target function E for a empty set of plans is zero. Specifically, $E(\emptyset, Q) = 0$. Since the set is empty and no plan can be used to improve the baseline method, the statement is true. Secondly, E is a nondecreasing function. Specifically, $E(S_1, Q) \leq E(S_2, Q)$ for all $S_1 \subset S_2 \subset P$. Since adding a new plan to a set of plans only increases the efficiency of the set, the statement is also true. Thirdly, and most importantly, E is a submodular function. This can be proved by the following theorem.

Theorem 3.4.2. *For all subset plans $S_1 \subset S_2 \subset P$ and plan $p \in P$ S_2, it holds that $E(S_1 \cup \{p\}, Q) - E(S_1, Q) \geq E(S_2 \cup \{p\}, Q) - E(S_2, Q)$, A set function E with this property is called submodular.*

Intuitively, if a plan is added to a small set of plans S_1, the efficiency is improved at least as much as if it is added to a larger set, where $S_2 \supset S_1$. Detail of the proof is in Appendix 7.3.

Based on the above discussion, we prove that our problem is an instance of the maximization of a monotone submodular function under a cardinality constraint problem. Theorem 3.4.3 presented in [15] shows that the greedy strategy can find a solution which achieves at least a constant fraction $(1 - 1/e)$ of the optimal efficiency.

Theorem 3.4.3. *If F is a submodular, nondecreasing set function and $F(\emptyset) = 0$, then the greedy algorithm finds a set S', such that $F(S') \leq (1 - 1/e) max_{|S|=K} F(S)$.*

In summary, our selection module works as follows: It first prunes candidate plans according to their space costs. Then, it evaluates efficiency of remained plans over a sample of queries and selects K plans step by step. Additional pruning heuristics can be used to prune candidate plans to be estimated. The detail of pruning heuristics can be found in Section

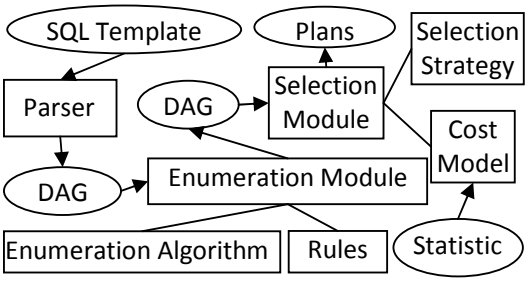


Figure 3.2: Architecture of our optimizer

3.5 Implementation Detail

In this section, we discuss the implementation detail of our query optimizer. The optimizer is based on Volcano query optimization framework. Figure 3.2 shows its overall architecture. It contains a parser, an enumeration module, and a selection module. Given a template, the parser parses it into a DAG structure. Given the DAG structure, the enumeration module systematically enumerates new plans by applying transformation rules to expand the DAG. The enumeration module contains a set of transformation rules and an enumeration algorithm, which controls how rules are applied. The output of the enumeration module is an expanded DAG, which contains all the plans with and without views. Given the expanded DAG, the selection module selects the most efficient plan set for the template. Specifically, the selection module contains a selection strategy, which determines how plans can be selected efficiently, and a cost model, which estimates efficiency of a candidate plan set based on statistics. Next, we discuss the detail of the enumeration module and the selection module.

3.5.1 Enumeration Algorithm

Algorithm 3.5.1 shows the enumeration algorithm. The input of the algorithm is the root node of a DAG, and a set of transformation rules. The output is the root node of the expanded DAG. The algorithm enumerates plans by applying rules to nodes in a recursive way. It first checks the base case (line 1-4). Next, it initializes a queue $ToExapnd$, which stores nodes that haven't been transformed, and a set $Created$, which stores all the created plans (line 5-6). Given a node, the algorithm first enumerates plans for its child (line 9-10). Then it applies rules to the node to derive alternative nodes (line 11-13). When an alternative node is created, the algorithm checks *created* to avoid duplication(line 14). Since additional new plans can be generated based on the new node, the new node is added into $Created$ and $ToExpand$. Transformation rules will be applied

to it in later iterations to generate new nodes.

Algorithm 2 Enumeration Algorithm

```
Enumerate(root, Rules)
 1: if root.expanded = true then
 2:     return root
 3: if root is a table-OR-Node then
 4:     return root
 5: Initialize a queue ToExpand with root.children
 6: Initialize a set Created with root.children
 7: while ToExpand is not empty do
 8:     AND-NODE node = ToExpand.pop();
 9:     for OR-NODE child in node do
10:         Enumerate(child)
11:     for Rule rule in Rules do
12:         if rule can be applied to node then
13:             newNode = rule.apply(node)
14:             if newNode not in created then
15:                 Created.add(newNode)
16:                 ToExpand.add(newNode);
17: root.children = created;
18: root.expanded = true
19: return root
```

3.5.2 Additional Transformation Rules

Our enumeration module is based on a set of transformation rules. The rule set contains rules that enumerate plans with views, and rules that enumerate plans with different execution orders and execution methods. The first set of rules are discussed in Section 3.3. We briefly describe the second set of rules.

A standard query optimizer can enumerate plans with different execution orders; but plans with different execution orders are partially enumerated due to the limits of online optimizing time. For instance, join orders are partially enumerated; and execution orders of other operations are determined with heuristics (e.g., an optimizer always push down a selection operation). Since some efficient plans may be based on the execution orders that are not enumerated, we add additional rules to a Volcano optimizer. Specifically, associative and commutative rules (e.g., $T_1 \bowtie T_2 \rightarrow T_2 \bowtie T_1$, $T_1 \bowtie T_2 \bowtie T_3 \rightarrow T_1 \bowtie (T_2 \bowtie T_3)$) are added to enumerate plans with all the join orders. Other additional rules, such as $(OP_u(T_l)) \bowtie T_r \rightarrow OP_u(T_l \bowtie T_r)$, $(\bowtie OP_u(T_r)) \rightarrow OP_u(T_l \bowtie T_r)$, and $OP'_u(OP_u(T_1)) \rightarrow OP_u(OP'_u(T_1))$ are added to enumerate orders of other operations. Note that there are an undetermined number of intersections in a template, the optimizer views the intersections as a single intersection group. It does not enumerate the order of intersections in the group. As we will discuss later, the sort-merge intersection method is always used. Thus, the order of

intersections in the group does not affect the efficiency.

A standard query optimizer can also enumerate execution methods for each operation in a plan. When enumerating execution methods, the optimizer considers whether indexes are available or not and whether tuples are ordered or not to prune inefficient methods. Our optimizer also use those heuristics. In addition, in our setting the optimizer can assume that some desired properties exist in tables or views, since tables and views can be stored as the optimizer wants. Specifically, we assume: 1. Tuples of a table or a view are sequentially stored on a disk. 2. Tuples are ordered according a global order. With those properties, the optimizer can select the most efficient method for an operation. For intersection, the sort-merge algorithm, which is also used in document search systems, is selected. For selection, the index-based selection is selected if a corresponding index is available. For projection, the scan method is used. For join, sort-merge join or index-join performs best in different cases, so both of them are enumerated.

3.5.3 Additional Selection Rules

Although the greedy algorithm is efficient, it still needs to evaluate the efficiency of all the candidate plans in each step. We propose several heuristic rules to prune the candidate plans to be evaluated.

First, the space constraint can be applied to prune candidates plans. Specifically, if views used in a subplan of the DAG structure are larger than the space budget B, then all the plans based on this subplan are pruned directly.

Second, a traditional query optimization task, the best plan is directly constructed by via selecting the most efficient plan of its children nodes recursively. Those plans that are based on inefficient sub-plans would be pruned directly. Similarly, when the optimizer selects the most efficient plan, it can select the most effluent subplan from its children.

Third, among candidate plans, a plan may be more efficient than another plan for all the queries, while it requires more spaces than the other. When the efficient one satisfies the space constraint, the inefficient one can be directly pruned.

Chapter 4

Experiment

4.1 Experiment Setting

To evaluate our proposed approach, we apply our approach to two different CTS systems. One is the EntitySearch task, and the other is TopicSearch task. The input and output of both systems are described in Section 1. Specifically, for a search system, we first apply our approach to derive an execution algorithm and indexes. Then, we built a prototype system based the execution algorithm and indexes. Finally, we evaluate the performance of the execution algorithm and compare with other baseline methods.

For each search task, we use our optimization framework to derive plans and views in three different settings. In the first setting, the optimizer outputs only one plan without any space constraint. We use $Plan@1$ to denote it. In the second setting, the optimizer outputs 3 plans without any space space constraint. We use $Plan@3$ to denote this setting. In the third setting, the optimizer outputs 3 plans with a space constraint. We use $Plan@S$ to denote this setting. Here, we set the space constraint as the size views used in a plan should be less than 4 times of the size of original data. In each setting, the final online execution algorithm always selects the most efficient plan from the derived plans to answer a query.

For each search task, we also compare the performance of the derived algorithm with a baseline method. The baseline algorithm is based on an inverted index, which maps a keyword to a list of documents associated with the keyword, and an forwarding index, which maps a document to a list of topics or entity instances in the document. The detail of the baseline algorithms for both search tasks can be found in 4.2 For the entity search system, we also implement the algorithm used in [7] as another baseline method.

Since we focus on search efficiency, we evaluate the performance as average query response time.

4.2 Baseline Algorithms

Algorithm 3 shows the baseline algorithm for the EntitySearch task. The algorithm is based on an inverted index, which maps a keyword to a list of documents associated with the keyword, and an forwarding index, which maps a document to a list of entity instances in the document. Specifically, From step 1 to 3, the algorithm first retrieves a list of relevant documents. DL, by intersecting inverted lists of each keyword. From step 4 and 5, the algorithm retrieve entities from each relevant document based on the forwarding index. From step 6 to 7, the algorithm determines whether a retrieved entity is the given type.

Algorithm 3 Baseline Algorithm for EntitySearch

Input: Query $K = \{k_1, k_i, \}$, Type $\#Type$
Output: $Score(E^J)$ for all entities E_j

1: **for** k_i in K **do**
2: fetch the hit list H_{k_i} for k_i
3: List $DL = H_{k_1} \cap H_{k_2}, ... \cap H_{k_i}$
4: **for** d_i in DL **do**
5: fetch $d^E = \{E^1, ...E^n\}$ from d_i with a forward index.
6: **for** E^j in d_i^E **do**
7: **if** E^j is $\#Type$ **then**
8: Update $Score(E^j)$

Algorithm 4 shows the baseline algorithm for the TopicSearch task. The baseline algorithm is similar to Algorithm 3. It is also based on an inverted index and a forward index.

Algorithm 4 Baseline Algorithm for TopicSearch

Input: Query $K = \{k_1, k_i, \}$, Time Period p
Output: $T^j\}$ for all topics T_j

1: **for** k_i in K **do**
2: fetch the hit list H_{k_i} for k_i
3: List $DL = H_{k_1} \cap H_{k_2}, ... \cap H_{k_i}$
4: **for** d_i in DL **do**
5: **if** $d_i \in p$ **then**
6: fetch $d^T = \{T^m, ...T^n\}$ from d_i with a forward index.
7: **for** T^j in d_i^T **do**
8: Update Score(T^j)

4.3 Entity Search System Result

We build a prototype EntitySearch system on two different data sets. Specifically, a small data set, denoted as $Set1$, contains 10000 web pages, and a large data set, containing 100000 web pages, denoted as $Set2$. In

both data sets, we annotate entity types and entity instances from web pages in a dictionary based approach. The dictionary contains 42 entity types, such as “computer software” and “computer scientist”, and 32900 different entity instances in computer related domains. We random select 1000 queries, such as “database, #computer scientist”, or “search, #computer software”, as testing queries to evaluate the performance of an algorithm.

4.3.1 Generated Plans

For each setting, we first use our optimizer to derive plans. Then, we implement an execution algorithm and its indexes based the derived plans.

Figure 4.2 shows the plans derived by our optimization framework for three settings. Specifically, Plan A is derived for the setting $Plan@1$. Plan A, B, and C are derived for the setting $Plan@3$. Plan D and E are derived for the setting $Plan@S$. We note that the optimizer outputs only two plans for $Plan@S$ even the K is 3. It is because there is no other plan that is more efficient than those two plans for any query. Next, we describe those plans and views in each plan.

Plan A computes a query in 4 steps. First, it scans a view, $View(\pi_{DID}(\sigma_{Key=k_i,type=t_i}(T_1 \bowtie T_2 \bowtie T_3))$ (denoted as V_1), to get a list of DIDs that are associated with a keyword k_i and contains entities of type t_i. Second, it computes a list of DIDs that contain all the keywords and entities of the given type by intersecting DID lists of different keywords. Third, it scans a view, $View(\pi_{DID,Entity}(\sigma_{Key=k_i,Type=t_i}(T_1 \bowtie T_3 \bowtie T_4))$ (denoted as V_2), to get a list of DID and Entity pairs associated with a keyword k_i and the type t_i. Forth, the plan uses the sort-merge join to join the DID list from the second step and the DID and Entity pair list from the third step to get the result.

Plan A uses two parameterized views. Specifically, V_1 represents a set of views, each of which materializes DIDs that are associated with a keyword, k_i, and contain entities for a type t_i. We can implement V_1 as a typed inverted index, where it uses a keyword k_i and a type t_i as the key to retrieve a list of associated documents. V_2 represents a set of views, each of which materializes DID and Entity pairs for a keyword k_i and a type t_i. We can implement V_2 as a contextualized inverted index. The index uses a keyword k_i and a type t_i as the key. An inverted list of a key is a contextualized list, which contains a list of DIDs and contextual information of each DID, specifically, Entities of the given type in each DID. This index can be viewed as a generalization of the neighborhood index in [2], where the inverted list contains a list of DIDs

and neighbor words of the given keyword in each DID.

Plan B computes a query in 2 steps. First, it scans V_2 to get a list of DID and Entity pairs of a keyword k_i and the type t_i for different keywords. It is similar to the third step in Plan A. However, Plan B access a set of views for different keywords. Second, it intersects the DID and Entity pair lists of different keyword and type pairs to compute the result. Plan B only uses V_2 to compute the query. Its implementation has been discussed in the previous paragraph.

Plan C computes a query in 3 steps. The first two steps are the same with those in Plan A. It computes a list of DIDs that contain all the keywords and entities of the given type. In the third step, Plan C computes the join operation with an index based join method. Specifically, for each document in the DID list, it accesses a view $View(\pi_{DID,Entity}(\sigma_{DID=d_i,Type=t_i}(T_3 \bowtie T_4))$ (denoted as V_3) to get entities that are in the document d_i and belong to the type t_i.

Plan C uses two parameterized views. One is V_1, which is discussed in Plan A. The other is V_3. It represents a set of views, each of which records entities that belong to a type t_i and are in a document d_i. We can implement it as a typed forward index. It uses t_i and d_i as a key. For each key, it records entities that belong to t_i and are in d_i.

Plan D compute a query in four steps. First, for each keyword k_i, it scans a view, $View(\pi_{DID}(\sigma_{Key=k_i}(T_1)$ (denoted as V_4), to get a list of DIDs associated with k_i. Second, it intersects DID lists for different keywords to get a list of DIDs associated with all the query keywords. Third, it scans a view, $View(\pi_{DID,Entity}$ $(\sigma_{Type=t_i}(T_3 \bowtie T_4)$ (denoted as V_5), to get a list of DID and Entity pairs of a type t_i. Fourth, it uses the sort-merge join method to join the DID list from the second step and the DID and Entity pair list from in the third step to get the result.

Plan D uses two parameterized view. Specifically, V_4 represents a set of views, each of which materializes DIDs associated with a keyword k_i. We can view V_5 as a traditional inverted index, which uses a keyword k_i as a key to retrieve a list of DIDs associated with the keyword. V_5 represents a set of views, each of which materializes DID and Entity pairs for a type t_i. It can be viewed as the entity index discussed in [6]. Specifically, the key of the index is a type t_i. The list associated with a key contains a list of DID and Entity pairs.

Plan E is similar to Plan C. The only difference is that Plan E scans V_4 instead of V_1 to retrieve a list of DIDs with a keyword k_i.

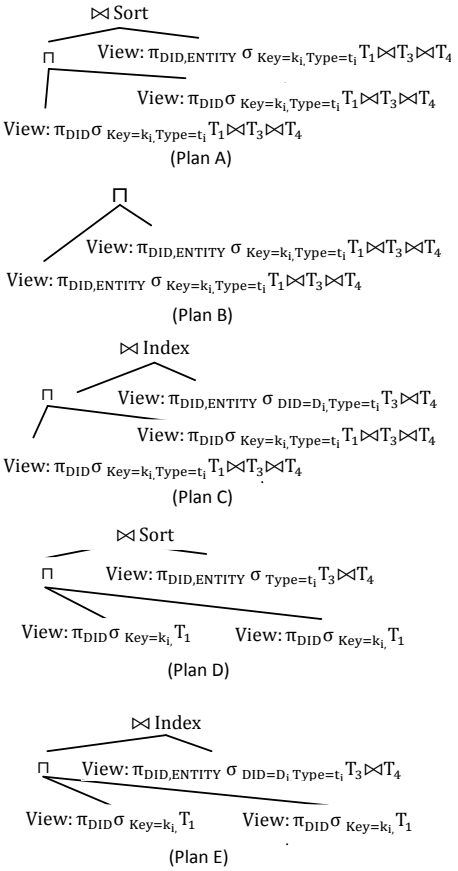


Table 4.1: Derived Plans for Entity Search

From the derived plans, we can see that our query optimization approach is able to derive efficient plans with views for efficiently computing the template. It can enumerate and output plans and indexes that are manually optimized for a system, such as entity index, or neighborhood index, in a principled way.

4.3.2 Efficiency of Derived Plans

Table 4.2: Average Query Performance

	Baseline	Entity	Plan@S	Plan@1	Plan@3
Set1	6928	5424	713	640	171
Set2	34663	15273	3132	2870	770

We evaluate the performance of each algorithm over testing queries. Table 4.2 shows the average query response time in term of millisecond over testing queries. From the results, we find that all the three algorithms can significantly improve the baseline algorithm. The algorithm *Plan*@3 is the most efficient. It is significantly better than *Plan*@1. It justify our assumption that a single plan can not answer every query

the most efficiently. From the results for different data sets, we find that although all the algorithms take a longer time to answer queries, *Plan*@3 can still answer queries within a reasonable time period.

Table 4.1 shows the space used by views used in each plan for $Set2$. The baseline method, which uses only an inverted index and a forward index, costs least space. *Plan*@1 and *Plan*@3, which improve the performance based on different views, cost around 20 30 times of the original space. It is an affordable cost considering that the gain of efficiency is large. $Plan@S$ generated by the optimizer with a space constraint, costs a small amount of additional space, while it still improves the performance. It shows that our optimizer can output efficient plans with different constraints, which control the tradeoff between space and efficiency.

Figure 4.1: Space Usage

Baseline	Entity	Plan@S	Plan@1	Plan@3
1.58GB	1.68GB	2.26GB	26.8GB	42.6 GB

4.4 Popular Topic Search System Results

We build a prototype TopicSearch system on a collection of twitter status. Specifically, the collection contains 16175673 tweets in 30 days. We use a dictionary based approach to annotate topics from tweets. Specifically, we use Wikipedia titles as our dictionary. It contains 6139418 titles and we identifies 93212 different topics in our collection. We random select 1000 queries including trending topics in Twitter (e.g., "Jennifer Lope") and Fortune 100 company names (e.g., WalMart) as testing queries to evaluate the performance of an algorithm. For this scenario, we do not list plans we generated due to the space limitation. We give the performance of each algorithm over testing queries in Figure **??**. From the results, we find that similar results as in EntitySearch scenario, all the three algorithms can improve the baseline algorithm. The algorithm *Plan*@3 is the most efficient.

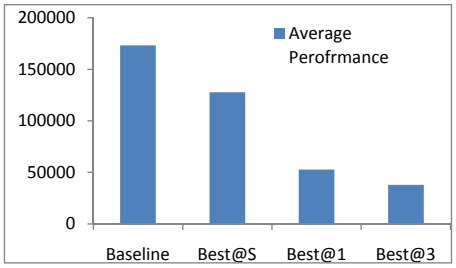


Figure 4.2: Average Query Performance

Experiments show that: (1) The derived algorithm and indexes significantly improve the efficiency the keyword-based baseline. (2) Our framework can automatically derive plans and indexes that are manually optimized for a system. (3) Our approach is capable to derive plans for different search systems.

Chapter 5

Related Work

We are now witnessing an emerging research trend on searching fine granularity units, such as entities, objects or topics from text data. Many systems have been proposed and studied [6, 2, 7, 16, 3, 17] for this kind of search tasks. Some of those works [7, 16, 3] almost exclusively focus on effectiveness of retrieve models. However, we focus on the efficiency aspect and improve the efficiency of those systems with a new execution algorithm with additional indexes.

Some new index structures have already been proposed and studied for those search systems [6, 2, 9]. For example, Cheng [6] builds an entity index for searching entities; and Cafarella [2] builds a neighborhood index for searching linguistic phrases. Those indexes are selected according to some heuristics and are task-dependent. In this paper, we propose a framework to automatically derive indexes and execution algorithms for those systems in a principled approach. By viewing those indexes as views, our framework will consider those proposed indexes when it enumerates different kinds of views that can be used.

Besides indexes, there are also existing other optimization techniques, such as caching [13] and sampling based intersection [17], to improve the efficiency of a search system. Such techniques are orthogonal to our problem and can be directly applied to a system, which is based on indexes and plans derived by our framework.

Our solution is most related with the works on query optimization in the database area. Our approach adopts a cost estimation based approach used in the traditional query optimization to select plans and views. However, a classical query optimizer only searches plans without materialized views [19, 10] for a single query. There are research works [4, 12, 14, 11, 8], which study the problem of using materialized views to optimizing queries. However, their solution solve the problem in a setting different from ours. Works in [4, 12] focus on exploring how views can be used in answering a query; and they assume a set of materialized views are given. In our problem views are not given; and our optimizer needs to enumerate all the possible views in order to find an optimal solution. Works in [11, 8] focus on solving the view selection problem,

which efficiently selects a subset of views that minimizes a cost function under constraints from a set of views. The solution explicitly assume views are given and they do not explore how a view can be used by a plan.

Our problem can also be viewed as an index configuration problem for a database. In the DB setting, some systems [1, 5, 18] have been proposed to solve this problem. The proposed systems enumerate different configurations with indexes or views for a workload and select the most efficient one. Their focuses are generating a small set of candidate indexes for optimizing the workload. Since the workload contains different kinds of queries, they do not fully explore all the useful views for each query. The most efficient configuration may not be enumerated. Our work innovates upon these works in a rather different setting: a complex text search system, where all the queries can be represented by a single template. With the concept of the template, our optimizer can extensively enumerate all the possible indexes(views) as well as plans for the template, and output a set of plans for the template.

Chapter 6

Conclusion

In this paper, we presented the dual-optimization framework, which optimizes execution plans and indexes at the same time for a search system. Extensive experiments show our optimization framework is able to derive efficient plans with indexes for different CTS systems.

Chapter 7

Additional Proof

7.1 Proof of Theorem 3.3.1

Before we prove the theorem, we first describe our assumptions and notations about a plan. We use the a tree structure to represent a plan, where leaves represent physical tables or views and internal nodes represents relation operators, denoted as op, or access operators, denoted as Acc. There are two access operators, which are the sequential access method and the random access method. The sequential access method, denoted as $Acc^S(T)$, sequential accesses the table T to fetch tuples of T into memory. The random access method, denoted as $Acc^R(Acc_l, Acc_r)$, takes two access operands as inputs. The left operand Acc_l fetches a set tuples used as keys, and the right operand Acc_r fetch tuples associated with each key. The execution order of a plan is determined by the tree structure.

To prove the theorem, we first prove the following lemma, which shows that our rules can generate plans with different access methods for a table T used in a plan P to compute a query Q.

Lemma 1. *Let P be an execution plan for a query Q, and uses the sequential access method $Acc^S(T)$ to access a table T. Rule 3 and Rule 4 are complete in generating different efficient accessing methods for T with materialized views.*

Proof. Let $Access^*(T)$ denote all the access methods to access T by a plan, and $Access(Acc^S(T))$ denote all the access methods generated by our rules based on $Acc^s(T)$ used in the plan. We prove $Access^*(T) = Access(Acc^S(T))$ by induction on the number of attributes in a table. Let T be any arbitrary table used in a plan and has N attributes. We assume that $Access^*(T') = Access(Acc^S(T'))$ is true for T' has less than N attributes.

Base case: In the basic case T only contains one attribute. $Access^*(T)$ contains the following methods:

1. $Acc^S(T)$: Obviously, $Acc^S(T) \in Access(Acc^S(T))$.

2. $Acc^S(T')$: Since $T'! = T$ and T' can replace T to answer Q correctly, $T' \subset T$ or $T \subset T'$. If $T \subset T'$, T' is larger than T, $Acc^(T')$ should be pruned from $Access^*(T)$. Since $T' \subset T$, $T' \in FilterView(T, Q)$. Rule 4 will generate $Acc^S(T')$ based on $Acc^S(T)$. Thus, $Acc^S(T') \in Access(Acc^S(T))$.

There is no random access methods for T. If the only attribute is used as the key in the random access method, the method will load all the tuples into the memory when it accesses all the possible keys. It doesn't need to access tuples with associated with each key. In this case it is equal to $Acc^S(T)$. If no attribute is used as key, it is also equal to $Acc^S(T)$. There is no random access method for a view V. Because V can correctly answer the query, V must have only one attribute as T. Similarly, we can prove there is no random access method for V.

In summary, for the base case, $Access^*(T) = Accsee(Acc^S(T))$.

Induction: In the general case, T contains N attributes. $Access^*(T)$ contains methods in the following forms:

1. $Acc^S(T)$: Obviously, $Acc^S(T) \in Access(T)$.

2. $Acc^S(T')$: Similar to the second method in the base case, we can prove $Acc^S(T') \in Access(Acc^S(T))$.

3. $Acc^R(Acc_l, Acc_r)$: Let V denote the tuples fetched by the random access method Acc^R. V must be T or a filter view of T. If $V = T$, Acc^R must use some attributes, denoted as Key, of T as keys to access tuples in T. To fetch all the possible keys, Acc_l must be in $Access^*(View_l^{Key})$, where $View_l^{Key} = \pi_{Key}(T)$. To fetch tuples with keys, Acc_r must be in $Acc_r \in Access^*(View^Key_r)$, where $View_r^{Key} = \pi_{Att(T)-Key}\sigma_{Key=K}(T)$. According to Rule 3, $Acc^R(Acc^S(View_l^{Key}), Acc^S(View_r^{Key}))$; $\forall Key \subset Att(T)$ are in $Access(T)$. Since both $Acc^S(View_l^{Key})$ and $Acc^S(View_r^{Key})$ have less than N attributes; by induction hypothesis, $Access^*(View_l^{Key}) = Access(Acc^S\ (View_l^{Key}))$ and $Access^*(View_r^{Key}) = Access(Acc^S\ (View_r^{Key}(K)))$. Thus, $Acc_l \in access(Acc^S(View_l^{Key}))$, $Acc_r \in Access(Acc^S(View_r^{Key}(K)))$, and $Acc^R(Acc_l, Acc_r) \in Access(Acc^S(T))$.

   If $V \in FilterView(T, Q)$, as discussed in the second form of this case, $Acc^S(V) \in Access(Acc^S(T))$. Similar to the case $V = T$, we can prove any random access method for V will be derived.

In summary, for a general case, $Access^*(T) = Access(Acc^S(T))$. □

Now we prove the theorem.

Proof. Let J be an execution plan for Q, and P be an arbitrary subtree of J. We use $S^*(P)$ to denote all the improved plans that are based on views and have the same execution order with P; and use $S(P)$ to denote all the plans generated by our rules. We prove that for any P our rules are complete. Since P is any subtree of J, if theorem holds for P, it also holds for J.

Since a plan is a tree structure, we prove the theorem using the structure induction technique. Assume that for any proper subtree P_s of P, $S(P_s) = S^*(P_s)$. There are several cases to consider.

Base case: P accesses a table node $Acc^(S)(T)$. We prove $S(P) = S^*(P)$ by contradiction. Let P' be a counter example, where $P' \in S^*(P)$ and $P' \notin S(P)$. P' is a plan, it must be one of the following forms:

- P' uses a different access operator to access the table. According to the lemma, all the efficient access methods are generated by our rules. Thus $P' \in S$.
- $P' = OP'(T')$. P' uses an addition relational operator. It has a different execution order with P. Since $S(P)$ only contains efficient plans that have the same execution order with P. Thus, $P' \notin S^*(P)$.

In summary, for the base case, $S(P) = S^*(P)$.

General Case 1: In a general case, P could be in a form of $op(P_s)$, where op is a unary operator, and P_s is a subtree structure. Let $S^*(P_s)$ be the set of all the efficient plans based on the execution order of P_s. The inductive hypothesis implies that $S(P_s) = S^*(P_s)$. According to our rules, $S(P) = \{op(p_{si})|\forall p_{si} \in S(P_s)\} \cup S(Acc^S(View(op(P_s))))$. $\{op(p_{si})|\forall p_{si} \in S(P_s)\}$ is constructed via directly using op and any sub plan of $S(P_s)$. $S(Acc^S(View(op_u(P_s))))$ is constructed by with Rule 1, 3 and 4. Specifically, Rule 1 derives a new plan $P'' = Acc^S(View(op_u(P_s))$, Rule 4 and 5 derives $S(P'')$. We prove $S(P) = S^*(P)$ by contradiction. Let P' be the counter example, $P' \notin S(P)$ and $P' \in S*(P)$. P' must be in one of the fellow forms:

- $P' = OP'(p'_s)$: OP' is different from OP and p'_s is a tree structure. Because OP' is different from OP, P' has a different execution order with P. According to the definition of $S^*(P)$, $P' \notin S^*(P)$.
- $P' = Acc^S(P'_s)$. It uses a sequential access method Acc to access a table. Let V' be a view storing all the tuples returned by the access method. $Acc^S(V')$ can also correctly answer the query, so V' is a filter view of $View(op(P_s))$. Since $Acc^S(View(op(P_s))) \in S(P)$, $Acc^S(V') \in S(P)$ according to Rule 4. Since $Acc^S(V') \in S(P)$, based on the lemma, Thus $P' \in S^(P))$.

- $P' = Acc^R(P_{sl}, P_{sr})$. As the previous case, we can prove $P' \in S^(P)$).

General Case 2: P is a plan with a K-nary operation node op operating on a set of subtrees. We can use the similar inferences in case 1 to prove our rules are complete for this case.

In all three cases, $S(P) = S^*(P)$. We conclude that our rules are complete. □

7.2 Proof of Theorem 3.4.1

Proof. To prove that the problem is a NP-hard problem, we reduce the K set cover problem to it.

Since the original problem is maximization problem, we covert it to an equivalent deterministic problem. The deterministic problem is whether there is a set of plans, S, which contains less K than plans and $E(S, Q)$ is larger than or equal to a predefined value C.

Given a set cover problem, which has a ground elements $U = \{u_1, u_2, ..., u_n\}$ and subsets $S_1, S_2, ..., S_k \subset U$, we map each element u_j to a query q_j, and each subset S_i to a plan p_i. If the element u_j is contained in S_i, then $e(p_i, q_j) = 1$; otherwise $e(p_i, q_j) = 0$. The original set cover problem can be reduced as whether there exists a best plan set such that $|S|$ is less than K and efficiency is equal to $|U|$.

If the original set cover problem has a cover of K sets, then there is a best plan set S that satisfies the constraint and $E(S, Q)$ is equal to $|U|$. To obtain the solution, we select K plans corresponding to K sets. Since every element is covered by at least one set, each query can be answered by at least one of selected plans. $E(S, Q) = \sum_{q_j \in Q} 1 = |Q| = |U|$.

If there is a best set S that satisfies the constraint and $E(S, Q)$ is equal to $|U|$, then there is a cover of K sets. We can construct the solution by selecting K sets corresponding to plans in the best set. Since $E(S, Q)$ is equal to $|U|$, which means every query can be answered by one plan in the set. Then every element will be covered by the set corresponding the plan. The K sets are a cover.

□

7.3 Proof of Theorem 3.4.2

Proof. Let $E(S, q_i) = \max_{p_j \in S} e(p_j, q_i)$ be the efficiency on a certain query q_i. From Eq. (3.8) we have $E(S, Q) = \frac{\sum_{q_i \in Q} E_i(S, q_i)}{|Q|}$. When a plan p is added into the plan set S_1, the efficiency for some queries increase while other queries are not affected. We denote the subscript set for these two kinds of queries as U_1 and

V_1, i.e., $U_1 = \{q_u | E(S_1 \cup \{p\}, q_u) > E(S_1, q_u)\}, V_1 = \{q_v | E_v(S_1 \cup \{p\}, q_v) = E_v(S_1, q_v)\}$. We do the same for S_2, and use U_2, V_2 as the subscript sets. Then we have $E(S_1 \cup \{p\}, Q) - E(S_1, Q) = \frac{\sum_{i \in U_1}(e(p,q_i) - E(S_1,q_i))}{|Q|}$ and $E(S_2 \cup \{p\}, Q) - E(S_2, Q) = \frac{\sum_{i \in U_2}(e(p,q_i) - E_i(S_2,q_i))}{|Q|}$. Since $S_1 \subset S_2$, it is not hard to see by definition that $U_1 \supset U_2$ and $E(S_1, q_i) \leq E(S_2, q_i)$. Thus we have

$$\begin{aligned} E(S_1 \cup \{p\}, Q) - E(S_1, Q) &= \frac{\sum_{i \in U_1}(e(p, q_i) - E(S_1, q_i))}{|Q|} \\ &\geq \frac{\sum_{i \in U_2}(e(p, q_i) - E(S_1, q_i))}{|Q|} \\ &\geq \frac{\sum_{i \in U_2}(e(p, q_i) - E(S_2, q_i))}{|Q|} \\ &= E(S_2 \cup \{p\}, Q) - E(S_2, Q) \end{aligned}$$

□

References

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[2] Michael J. Cafarella and Oren Etzioni. A search engine for natural language applications. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 442–452, 2005.

[3] Yunbo Cao, Jingjing Liu, Shenghua Bao, and Hang Li. Research on expert search at enterprise track of trec 2005. In *TREC*, 2005.

[4] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 190–200, 1995.

[5] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 146–155, 1997.

[6] Tao Cheng and Kevin Chen-Chuan Chang. Beyond pages: supporting efficient, scalable entity search with dual-inversion index. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 15–26, 2010.

[7] Tao Cheng, Xifeng Yan, and Kevin Chen-Chuan Chang. Entityrank: searching entities directly and holistically. In *VLDB '07*, pages 387–398, 2007.

[8] Rada Chirkova and Chen Li. Materializing views with minimal size to answer queries. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '03, pages 38–48, 2003.

[9] Junghoo Cho and Sridhar Rajagopalan. A fast regular expression indexing engine. In *In Proceedings of the 18th International Conference on Data Engineering*, 2001.

[10] G. Graefe and W.J. McKenna. The volcano optimizer generator: extensibility and efficient search. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 209 –218, April 1993.

[11] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. on Knowl. and Data Eng.*, 17:24–43, January 2005.

[12] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering queries using views (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 95–104, 1995.

[13] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. *World Wide Web*, 9:369–395, December 2006.

[14] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Rec.*, 30:307–318, May 2001.

[15] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions. *Mathematical Programming*, 14:265–294, 1978. 10.1007/BF01588971.

[16] Zaiqing Nie, Yunxiao Ma, Shuming Shi, Ji-Rong Wen, and Wei-Ying Ma. Web object retrieval. In *WWW '07*, pages 81–90, 2007.

[17] Alkis Simitsis, Akanksha Baid, Yannis Sismanis, and Berthold Reinwald. Multidimensional content exploration. *PVLDB*, 1(1):660–671, 2008.

[18] Alan Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the 16th International Conference on Data Engineering*, pages 101–, 2000.

[19] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 1st edition, 2001.

[20] Minji Wu and Amélie Marian. A framework for corroborating answers from multiple web sources. *Inf. Syst.*, 36:431–449, April 2011.