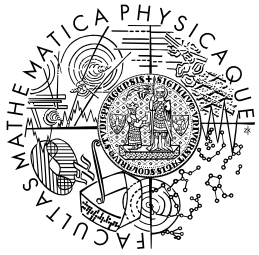

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Filip Mihál'

An application of AI methods for refining the storage strategy in multi-model database systems: A survey

Department of Software Engineering

Supervisor of the bachelor thesis: Ing. Pavel Koupil, Ph.D

Study programme: Computer Science

Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, Ing. Pavel Koupil, Ph.D, for his support, guidance, and productive discussions that led to the development of this thesis. I would also like to thank my consultant, Ing. Jáchym Bártík, for his valuable insights. Last but not least, I would like to thank my family and friends for their support throughout the course of my studies.

Title: An application of AI methods for refining the storage strategy in multi-model database systems: A survey

Author: Filip Mihál'

Department: Department of Software Engineering

Supervisor: Ing. Pavel Koupil, Ph.D, Department of Software Engineering

Abstract: Multi-Model database systems combine the advantages of traditional and NoSQL database systems. However, the management of these systems is challenging, as users have to design an appropriate storage strategy for their data. One of the most influential factors in the storage strategy is the selection of indexes. Indexes can significantly improve query performance, but they require additional storage space and maintenance overhead. Index selection problem is well-studied in the context of single-model Database Management Systems (DBMSs), but there is a lack of research in the context of multi-model database systems.

We address this problem by conducting a survey of current state-of-the-art index selection algorithms and evaluating their applicability to other DBMSs. The results reveal the strengths and weaknesses of existing algorithms and highlight the need for specialized algorithms for multi-model database systems. Moreover, we formulate open questions and suggest future research directions in this field. Our research provides a foundation for the development of efficient index selection algorithms for multi-model DBMSs.

Keywords: Multi-Model Data, Index Selection, Database Management Systems, Reinforcement Learning, Survey

Contents

Introduction **2**

1 Review of popular database systems **4**
1.1 Use of indexes in database systems 4
1.2 Relational database systems 4
1.3 Graph database systems 5
1.4 Key-value database systems 6
1.5 Document database systems 7
1.6 Columnar database systems 8
1.7 Multi-Model data 8
1.8 Comparison of selected database systems 9

2 Index selection problem **12**
2.1 Definitions & Notation 12

3 Index selection algorithms **14**
3.1 General formulation of ISP algorithms 14
3.2 Linear Programming based algorithms 15
3.2.1 CoPhy 15
3.2.2 Summary 18
3.3 Greedy algorithms 18
3.3.1 Drop heuristics 19
3.3.2 AutoAdmin 20
3.3.3 Anytime Database Tuning Advisor 23
3.3.4 Extend algorithm 23
3.4 Reinforcement Learning based algorithms 25
3.4.1 NoDBA 25
3.4.2 Budget-aware RL algorithm 27
3.5 Comparison of the algorithms 28

4 Open questions and challenges **33**

Conclusion **35**

Bibliography **36**

List of Figures **40**

List of Tables **41**

List of Abbreviations **42**

Introduction

The way we store and access data has been changing rapidly over the past few decades. At first, data was stored in traditional single-node *relational databases*, which were designed to handle *normalized* and *structured data*. Developers had to first define the *schema* of the data and then store it in a structured manner. However, as the amount of data grew, these databases became too slow and expensive to scale. Additionally, the need to store *denormalized* and *unstructured data*, such as logs and multimedia data, emerged. This led to the development of *NoSQL databases*, which were designed to handle unstructured data and scaled very well. Parallel to this, many new data models and their underlying DBMSs emerged. They were designed to handle specific types of data and were often more efficient at handling that data than other DBMSs.

In recent years, the amount of data generated by various applications and devices has grown exponentially, resulting in the need for even more advanced and sophisticated database systems. If developers want to handle their data efficiently, they need to combine different data models in a single system, i.e., they have to deal with so-called *multi-model data*. Therefore, *multi-model DBMSs* have been emerging in recent years.

Multi-model databases are designed to support multiple data models against a single, integrated backend. Their benefits include storing and accessing data of different types and consolidating cross-model operations. Modern organizations are in need of multi-model databases as they process a large amount of data from various sources and several forms. With these benefits, however, comes the challenge of designing an appropriate storage strategy. Vigorous research has been conducted in the field of multi-model databases to find the optimal storage strategy for given multi-model data [1, 2, 3, 4].

Although choosing the right combination of data models is critical to database performance, it is not the only factor affecting the performance of DBMS. Index and materialized views selection are other important factors for optimizing database performance [5]. Index selection involves creating the appropriate indexes on database attributes to speed up queries and improve performance. Materialized view selection means precomputing frequently executed queries to eliminate repetitive computation. Other essential management methods include database partitioning, which involves dividing a database into smaller, more manageable sections, and query optimization, which involves reorganizing queries to improve performance. Using these management methods, organizations can optimize database performance to meet the demands of their modern workflows.

This thesis focuses solely on index selection and its applicability to multi-model data. It is a well-studied problem in the field of single-model databases [6, 7, 8, 9, 10, 11] and its influence on the DBMS performance is significant.

Index selection used to be managed by database administrators and required full-time attention to maintain a satisfactory database state. There has been a great effort to automate the index selection in recent years. Various index selection algorithms have been developed, and their performance has been constantly improving [6].

With the rise of multi-model databases, the index selection problem has be-

come even more complex and requires even more human involvement. We believe that MMDBMS management needs a new level of abstraction to optimally select indexes to maximize its performance.

The main objectives of this thesis can be summarized as follows:

- *Analysis of indexing in various DBMSs* First, we analyze the most popular DBMSs, describe their use cases, and compare their typical workloads. Next, we conduct a comprehensive analysis of indexing strategies these DBMSs use. We mainly focus on the index types they support and the data structures they use to implement these indexes.

- *Analysis of existing approaches* As index selection is already a well-studied problem in the context of relational databases [6], we first analyze the current state-of-the-art algorithms and select the ones that might be suitable for multi-model data. The algorithms we study vary in their approach, implementation, and complexity. We also consider machine learning-based algorithms, which are currently inferior to the other algorithms, but are extremely promising for the future development of index selection algorithms [5].

- *Verification of their broader applicability* We consider the applicability of the selected algorithms to other DBMSs. We also compare the algorithms and suggest the most promising ones for each database system. In case the algorithms are not applicable, we analyze the reasons and formulate ideas for their improvement.

- *Definition of open questions* Finally, we formulate open questions and challenges for future research of index selection in multi-model DBMSs.

Outline First, in Chapter 1, we explain popular data models and their underlying DBMSs. We compare the data models based on their real-world applications, typical queries, advantages, and disadvantages. We also compare the DBMSs based on their support for different index types. Then, we describe the modern methods for working with multi-model data and explore the differences between them. Next, in chapter 2, we describe the index selection problem and its complexity. We also introduce theoretical concepts that are essential for understanding index selection algorithms in a broader context. In Chapter 3, we describe the current state-of-the-art index selection algorithms and group them according to their approach, namely machine learning, integer linear programming, and greedy algorithms. we then compare the selected algorithms based on their applicability to other data models, the number of supported index types, and the complexity of the algorithm. Finally, in Chapter 4, we formulate challenges and open questions for future work in the field of multi-model DBMSs and index selection.

1. Review of popular database systems

There exist hundreds of DBMS [12]. Each serves a variety of purposes and is suitable for different data. If we want to apply the index selection algorithms to multi-model DBMSs, we need to understand the database systems, supported data models, their characteristics, and how indexes are used to optimize query performance.

This chapter will introduce the most popular data models used in multi-model and NoSQL database systems. In particular, we will describe relational, graph, key-value, document, and columnar models. For each model, we will provide a brief overview of real-world applications, typical queries, and how they implement indexes. We will also select a representative DBMS for each model and specifically describe its index implementation. Finally, we will compare the different DBMSs in the context of the data they work with and the index types they use.

1.1 Use of indexes in database systems

A database index is a data structure used to quickly locate data without searching unnecessarily large portions of the database. This benefit comes at the cost of additional write complexity and storage space to maintain the index data structure. Most modern database systems implement indexes to let developers speed up their queries and manage their data as efficiently as possible. As popular DBMSs use different data models, the indexes they use differ as well. Some DBMSs use them to quickly find relationships between entities, some to find specific values instantly, and others to enforce strict constraints. The most common use cases for indexes include fast lookups, data aggregation, sorting, and range queries.

1.2 Relational database systems

The relational model is the most popular data model in the world [12]. It is ideal for storing normalized and structured data, where information can be organized into tables with columns and rows. This type of data, for example, includes financial records, inventory lists, customer information, and more. More generally, relational databases are ideal for keeping the data consistent (i.e., ACID transactions are supported) and well-organized.

Typical queries in the relational model involve selecting specific data from one or more tables, joining multiple tables, filtering data based on certain criteria, grouping, and aggregating data. Various types of indexes are used for this purpose, including B-trees, hash indexes, and bitmap indexes [13]. Namely, B-trees are used for range queries and sorting data, hash indexes for fast lookups in large tables, and bitmap indexes for handling data with boolean operations.

Three representative DBMSs that support the relational model are, e.g.,

MySQL[1], Oracle[2], and PostgreSQL[3].

Indexes in PostgreSQL PostgreSQL provides several indexes, each with its own characteristics and use cases. In particular, it supports B-tree, hash, GiST, GIN, and SP-GiST indexes [14]. B-tree is the default index type used for indexing simple data types, such as integers, floats, and timestamps. It is well-suited for range queries and sorting data. The Hash index is used for indexing simple equality queries on fixed-length data types such as integers. It is implemented as a hash table, allowing fast lookups for exact matches. However, it does not support range queries and sorting. GiST index is used for indexing geometric and text data types and implementing full-text search. GIN index is used for indexing arrays and composite types, as well as for implementing full-text search. Last but not least, SP-GiST index is used for efficient range queries on multidimensional data types such as geographic coordinates and time-series data.

1.3 Graph database systems

The graph database system is a particular NoSQL database designed to store and manage data in the form of nodes and edges (i.e., graph), allowing for efficient management and querying of complex relationships. Graph databases excel at managing and querying data that has complex relationships, such as social networks, recommendation engines, and fraud detection systems. They are also useful for storing data that is constantly changing and evolving.

Typical queries in a graph database include finding the shortest path between two nodes, identifying patterns in the data, and recommending related nodes based on common attributes. The advantages of graph databases include high performance for complex queries and the ability to add or modify relationships and properties without changing the entire schema. In graph databases, the relationships are not calculated at query time but are stored as physical connections between nodes, allowing for faster and more efficient queries that traverse the relationships between nodes without the need for complex join operations.

The design of graph databases offers a great selection of various indexes [15]. Indexes can be created on node or edge attributes. They can be created on a single property or multiple properties. Lastly, they can be optimized for range, exact match, or text queries. Hence, multiple data structures are needed to support these index types. Graph databases usually use B+ trees, hash tables, and inverted indexes.

The most popular DBMS implementing the graph model is Neo4j[4].

Indexes in Neo4j Neo4j supports range, lookup, text, point, and full-text indexes [15]. It automatically creates lookup indexes for node and edge labels to eliminate the need to search through all entities when executing a query. Figure 1.1 illustrates the possible index types in Neo4j. As we can see, only the

[1] https://www.mysql.com/
[2] https://www.oracle.com/database/
[3] https://www.postgresql.org/
[4] https://neo4j.com/

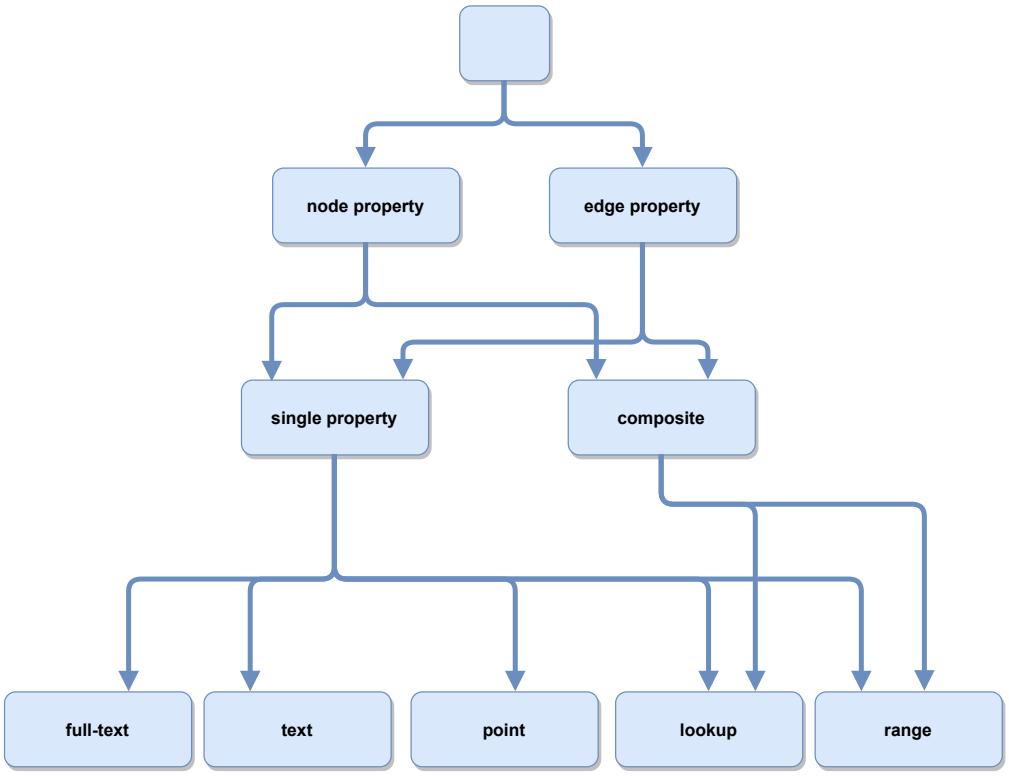

Figure 1.1: Possible index properties in Neo4j

lookup and range indexes can be created on multiple properties. The range index is the most flexible type. It can be used for exact match, range, and sorting queries. Point indexes only solve predicates operating on points, such as distance or bounding box queries. The text index is optimized for predicates based on prefixes, suffixes, and other text operations. Full-text index, as the name suggests, is used for full-text search.

1.4 Key-value database systems

The key-value model is the simplest model for storing data. Each piece of data is stored as a $(key, value)$ pair, with the key acting as a unique identifier and the value containing the actual data. It is used for storing semi-structured or unstructured data, such as user profiles, session data, or shopping cart data. The advantages of key-value models include faster read and write operations (e.g., in-memory), scalability, and the ability to handle large volumes of data with low latency.

Typical queries in the key-value model involve retrieving specific values based on their keys, updating or deleting values, and performing basic operations such as incrementing or decrementing values.

Each key in a key-value DBMS is like a primary key in a relational database system. In this case, keys are represented as hash tables, allowing fast retrieval of stored values. In other data models, to query attributes other than the primary key, users need a secondary index. In the case of key/value model, the values are typically considered as black boxes, and the only way to query them is by

assigned keys. The key lookups are usually instantaneous, hence there is no or very little need for secondary indexes. However, some key-value databases do support secondary indexes, but users must explicitly define them [16].

The most popular representatives of key-value DBMSs are Redis[5] and Riak[6].

Indexes in Redis As opposed to other key-value databases, in Redis, values can contain more complex data types, such as lists, geospatial, and sorted sets. These values are then more likely to be queried on attributes other than their keys. Hence, the need for secondary indexes arises. Redis uses hash tables for primary indexes and sorted sets for secondary indexes. There is no automatic way to create a secondary index in Redis; the user must explicitly define it by implementing a new sorted set of the indexed attributes [16]. The sorted list then maps each attribute to a set of primary keys whose values match the attribute.

1.5 Document database systems

The document model represents data in hierarchically structured aggregates (documents), which are self-contained units of information. DBMSs that use the document model are ideal for storing unstructured or semi-structured data, such as JavaScript Object Notation (JSON) [17] or eXtensible Markup Language (XML) [18] formats. In the industry, document DBMSs are used for content management, social media, and internet of things (IoT) applications. The document model allows for flexible schema design and can easily accommodate changes in data structure over time [19].

Typical queries in the document model involve selecting specific documents based on their fields, filtering data based on specific values, and sorting documents based on certain criteria. The advantages of these queries include the ability to handle complex and dynamic data, faster read and write operations, and theoretically unlimited scalability.

The most common index types used in the document-oriented DBMSs are single field, compound, and multi-key indexes [20, 21]. In particular, single field indexes are used to exact match queries on a specific field and compound indexes to query on multiple fields. Both single field and compound indexes are represented as B-trees. On the other hand, Hashed indexes are used to look up data in large collections efficiently and are represented as hash tables. Finally, document-oriented DBMSs also utilize full-text indexes (e.g., inverted index), geospatial indexes, and multi-key indexes.

Two representative document-oriented DBMSs are MongoDB[7] and Couchbase[8].

Indexes in MongoDB MongoDB is a distributed database system, therefore the data is stored across multiple nodes. This allows MongoDB to scale horizontally and handle large amounts of data and traffic. As for indexes, several types

[5] https://redis.io/
[6] https://riak.com/
[7] https://www.mongodb.com/
[8] https://www.couchbase.com/

are supported, including single field, compound, multi-key, geospatial, text, and hashed indexes [20]. Moreover, hashed indexes support sharding using shard keys to determine how the data is partitioned across the cluster [22].

1.6 Columnar database systems

Unlike traditional row-based (relational) models, the columnar model represents data by column. Rows then correspond to a collection of columns, and tables (also known as column families) are represented as a collection of rows. Thus, each row does not need to have the same set of columns as opposed to the relational model. The columnar database systems are well-suited for storing large amounts of structured data, such as financial data or log files.

Typical queries in the columnar database systems involve selecting specific columns and filtering data based on specific values. The advantages of using the columnar DBMSs for these queries include faster read and write operations, improved compression and data storage, and the ability to scale horizontally by adding more servers.

As systems that use the column model are usually distributed databases, their index architecture differs from other single-node databases. Primary indexes are comprised of a partition key and a clustering key. The partition key determines which node the data is stored on, while the clustering key determines the order of data within a partition. Secondary indexes are hidden tables whose primary key is the indexed column and whose other columns are the primary key of the original table and some metadata depending on the index type. Secondary indexes are co-located with the source data on the same nodes [23]. The partition keys are represented as hashed tokens, and the indexed column values are stored in B+ trees.

The most popular representatives of columnar database systems are Apache Cassandra[9] and HBase[10].

Indexes in Cassandra As Cassandra is heavily optimized for write operations, it indexes data using Log-Structured Merge Trees (LSMT) [24]. LSMT might not be as efficient as a B+ tree for read operations, but it can write data in constant time [25]. Cassandra does not support multi-column indexes, and indexed columns are not supposed to have high cardinality [26, 27]. Hence, secondary indexes are not that common in Cassandra. It is more common to modify the database schema to make it more suitable for the queries being executed.

1.7 Multi-Model data

Multi-model data combines variously logically represented data into a single dataset. Multi-model data is becoming increasingly common as organizations generate and store large volumes of diverse data types. For instance, a social media platform may need to store user profiles using a relational model, the relationships between users using a graph model, and user-generated content using

[9] https://cassandra.apache.org/
[10] https://hbase.apache.org/

a document model. There are generally two approaches to storing multi-model data: *polyglot persistence* [28] and *multi-model databases* [29].

Polyglot persistence Polyglot persistence is an approach that involves using multiple database systems to store multi-model data [28]. These databases are then managed by database administrators, who are responsible for ensuring that the data is stored and managed correctly. The advantages of polyglot persistence include the ability to use the best-suited storage technology for each type of data, improved scalability, and better performance. On the one hand, it requires additional effort and complexity in the design and management of the system. It may also increase development and maintenance costs.

Multi-model database systems Multi-model DBMSs are designed to support multiple data models against a single, integrated backend [29]. The system can automatically process multi-model data without the need for manual intervention. Data is consistent and can be queried by a single (possibly extended) query language that supports querying across all involved data models. One of the disadvantages of multi-model databases is that they are complex to design and implement. Similarly to NoSQL systems, multi-model DBMSs are also quite immature compared to traditional database systems, which means that they are still evolving and may miss some features that are available in relational systems.

Two representatives of multi-model DBMSs are, e.g., ArangoDB[11] and OrientDB[12].

Indexes in multi-model databases In a multi-model database system, indexes work similarly to how they work in single-model databases. Based on our observation, the most significant difference is that MMDBMS must support a broader range of index types than single-model databases. The multi-model system must support the main index types depending on the supported data models to ensure efficient query processing. Another difference is that multi-model databases require more complex query optimization to determine which secondary index to use for a given query.

1.8 Comparison of selected database systems

Table 1.8 provides a comparison of selected popular DBMSs. Each data model is represented by a single representative. As we can see, most of the modern database systems are built to store semi-structured and unstructured data. This trend is caused by the large volumes of data that is unfeasible to store in a structured form. Relational (PostgreSQL) and columnar (Cassandra) DBMSs are on the other hand designed to store structured data.

When it comes to indexing strategies, the DBMSs optimize for fast read operations with the exception of Cassandra, which takes into account write operations as well. The most common index types are range and lookup indexes. They are

[11] https://www.arangodb.com/
[12] http://orientdb.org/

Table 1.1: Comparison of different DBMSs

	PostgreSQL[1]	**Neo4j**	**Redis**[1]	**MongoDB**[1]	**Cassandra**
Model	Relational	Graph	Key-value	Document	Columnar
Data type	Structured	Semi-structured, unstructured	Semi-structured, unstructured	Semi-structured, unstructured	Structured
Indexed attribute	Columns	Node/edge properties	Keys	Fields	Columns
Range indexes	B-tree, SP-GiST	B+ tree	Sorted set	B-tree	LSMT
Fast lookups	Hash, B-tree	Hash, B+ tree	Hash	Hash, B-tree	Hash
Composite indexes	Yes	Yes	No[2]	Yes	No
Text indexes	Yes	Yes	No	Yes	Yes
Other indexes	GiST, BRIN	Point	No	Multi-key, Geospatial	No

[1] This DBMS supports multiple models; in our case, we consider it to represent the selected data model.
[2] Neither secondary indexes nor composite indexes are supported in Redis. However, we can construct composite indexes using sorted sets.

usually implemented using B-trees and hash tables, respectively. MongoDB, PostgreSQL, and Neo4j support various index types ranging from geo-spatial to array indexes. With a proper index selection, these DBMSs can efficiently process a wide range of queries, but the complexity of the index selection process increases. Another pattern that we can observe is the limited or non-existent support for secondary indexes in columnar and key-value DBMSs. For these DBMSs, it is recommended to modify the database schema instead of using secondary indexes.

Generally, the DBMSs share common indexing strategies, but they differ in the implementation details and supported index types.

2. Index selection problem

A proper selection of database indexes, referred as index selection problem (ISP), can have a significant impact on the performance of the database system. When chosen and used correctly, indexes can speed up query execution [6]. However, selecting the right indexes requires careful consideration of the data model, query patterns, and application workload. Adding too many indexes can slow down write performance and increase storage requirements, while not having enough indexes can lead to slow query performance. Therefore, it is important to carefully evaluate the trade-offs between the performance benefits of indexes and their associated costs in terms of storage and write performance. Much effort has been put into automating the index selection process [9, 30, 8, 7, 10, 11].

However, the index selection is an NP-complete problem [31]. Finding the optimal solution is, therefore, computationally expensive, and algorithms must balance the solution's quality and the time needed to find it. Not only is the set of possible index combinations exponential, but indexes interact with each other, so the benefit of one index is affected by the presence of another one [32]. This makes the problem even more complex. Additionally, the benefit of an index can only be computed by calling the database API, which is a time-consuming operation. Despite these challenges, many algorithms achieve commendable results. We will describe a selection of them in the next chapter.

2.1 Definitions & Notation

In this section, we introduce the basic definitions and notation that will be used throughout the thesis. There are many different notations used in the literature, therefore we use the notations that are most common in the papers we refer to.

Workload The set of queries that are executed on the database. Workload will be denoted as W throughout the thesis.

Constrains In the context of index selection, constrains are the stop conditions for the algorithms. They are usually defined as a storage budget, time limit or the number of indexes. We denote a set of constrains as K.

Index candidate An index that is considered for the index selection. It is usually picked from the set of all possible indexes by a heuristic that is applied to the workload. A set of candidates which we denote as I is usually a subset of the set of all possible indexes. Most of the time I is too big to satisfy the constrains of the algorithms.

Configuration A set of index candidates, usually denoted as C. It represents a possible index selection.

Cost A measure of the performance of a query. We will use the term cost to refer to the execution time of a query. It is a function of the query and the

configuration. The units of the cost are usually milliseconds but it can be any other unit. The cost for a query q and a configuration C is denoted as $cost(q, C)$.

what-if call A call to the database optimizer that asks for the cost of a query with a given configuration. The so-called virtual indexes are created and the optimizer uses them to estimate the cost of the query.

Workload Monotonicity Let S_1 be an arbitrary index set and S_2 be a subset of S_1. Then for an arbitrary query q from the workload, it holds $cost(q, S_1) \leq cost(q, S_2)$.

Offline index selection The index selection is executed on presumably fixed workload and database.

Online index selection The index selection is executed on a changing workload and database. It creates and drops indexes on the fly. Additionaly, it can base its decisions on the past experience.

Query optimizer A component of the database management system (DBMS) that is responsible for the query execution plan selection. It can approximate the cost of executing a query with a given configuration.

Data partitioning A technique that is used to distribute the data across multiple machines in a cluster.

Inverted index An index that maps a value to a set of records that contain the value. It is mostly used for efficient full-text search in text-based documents. The implementation of an inverted index involves parsing and tokenizing each document and creating a term-to-document mapping.

Horizontal Scaling A technique that is used to increase the performance of a database by adding more machines to the cluster. Each machine then handles a subset of the overall workload, potentially providing better efficiency than a single high-speed high-capacity server.

3. Index selection algorithms

After it had been proved that the problem is NP-complete [31], further effort in this area was mostly shifted toward heuristic algorithms rather than algorithms attempting to find the optimal solution [30, 8, 7, 33, 10, 11]. As a consequence, modern index selection algorithms approximate the solution. However, it is also hard to approximate the optimal solution [34]. Hence, the index selection problem is continuously being researched and new algorithms are being developed.

3.1 General formulation of ISP algorithms

Despite the great variety of index selection algorithms, most of them share some key traits. As illustrated in Figure 3.1, we can construct a general architecture that will help us understand the main concepts and will be used to describe the algorithms in the following chapters.

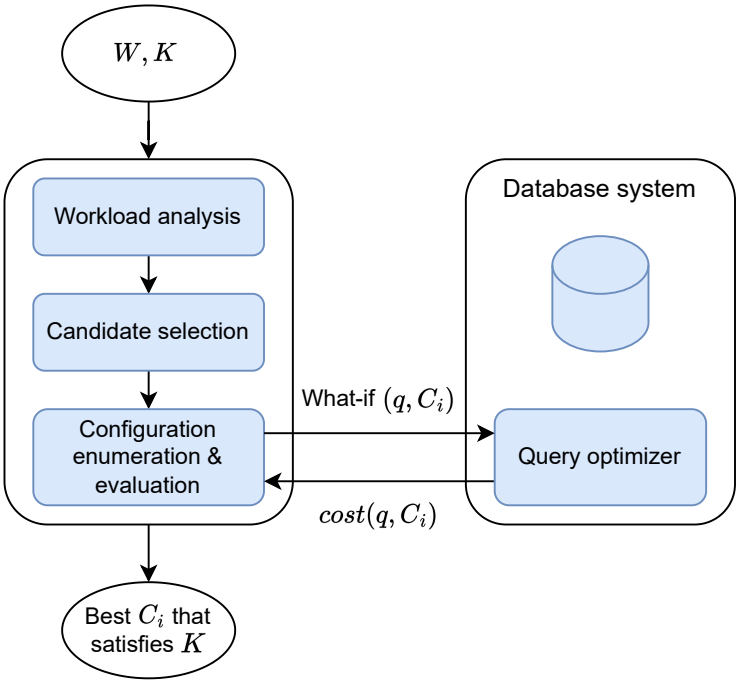

Figure 3.1: General architecture of the index selection algorithms

As the efficient use of indexes depends on the queries executed on the database, the queries are the main input of the *index selection algorithms*. The set of input queries is called the *workload*. Additionally, the database systems that require ISP algorithms are usually robust, and indexing all attributes is not feasible. Therefore, the algorithms must also be able to handle *constraints*. There are multiple ways to constrain the index selection process. The most common constraints are, for instance, the number of indexes, the index storage space, and the algorithm execution time.

ISP algorithms then process the workload and analyze its contents. Based on the analysis, they select a set of indexes that might improve the workload's performance. This set of indexes is called the *candidate set*. The candidates form sets

called *configurations*. A configuration is a subset of the candidate set that satisfies the given constraints. It represents a possible solution to the index selection problem. The configurations are then evaluated, and the best one is selected as the final result. As enumerating through all the possible configurations is usually not possible, the algorithms use *heuristics* to select the best configuration.

Heuristics differ based on the given constraints. Usually, the evaluation is done by estimating the cost of the workload execution with the selected configuration. In order to save time and not alter the state of the database, the algorithms usually use *what-if* calls to the *query optimizer* to estimate the execution cost of a query for the given configuration.

Although most algorithms use the same general architecture, they differ in many aspects. The following sections introduce three groups of algorithms that solve the index selection problem. Each group has a few selected representatives that will be described in detail.

3.2 Linear Programming based algorithms

The main idea of the index selection problem is to minimize the query execution cost by implementing appropriate indexes that satisfy the given constraints.

Let $q \in W$ where W represents the workload. Let I be the set of index candidates. The cost of the query q for a configuration $C \subseteq I$ is $cost(q, C)$. Weight f_q represents the frequency of the query q in the workload. To find the optimal solution, the algorithm must consider all constraint-satisfying subsets of the index candidates I and return the set whose cumulative cost is the lowest.

$$\arg\min_{C}\{\sum_{q \in W} f_q cost(q, C) | C \subseteq I\}$$

Constraints such as the storage budget and the number of indexes can be easily formulated by a set of linear inequalities. By selecting proper variables, the whole model can be then interpreted as a linear program.

3.2.1 CoPhy

A representative of algorithms that solve the index selection problem by linear programming is CoPhy [9]. CoPhy's core is based on a binary integer program (BIP). It works with multi-column indexes and allows multiple indexes per query. CoPhy is originally built to be constrained by the storage space but the LP nature allows us to use any kind of linear constraints. Index candidates are selected for each query and then merged into a single set. The main difference between CoPhy and other LP algorithms is the way it constructs its variables.

CoPhy considers configurations in the context of a query, i.e., multiple configurations are created for each query. In order to minimize the number of what-if calls, configurations are set to be atomic. This means that each configuration contains only one or zero indexes per table. The set of configurations for a query q is denoted as A_q. Index candidates that belong to the same table T_j are grouped together into a single set S_j. Binary variables $x_{q,C}$ indicate whether the given configuration is used for the query q. Variables $y_{q,j,s,C}$ ensure the atomicity of the configurations. Binary variables z_i model whether a specific index $i \in I$ is used.

The program constraints ensure that the following four conditions are met: (1) each query must use a single C. (2) C must be an atomic configuration. (3) Each z_i that is included in a used C must be set to 1, and (4) the storage constraint must be satisfied as well. The size of an index i is denoted by $size(i)$. The storage budget is denoted as B. The model is represented by the equation illustrated in Figure 3.2.

minimize:

$$\sum_{q \in W} \sum_{C \in A_q} x_{q,C} \cdot cost(q, C)$$

subject to:

$$\begin{aligned}
\sum_{C \subseteq I} x_{q,C} &= 1 && \forall q \in W \\
\sum_{i \in S_j \cup I_\emptyset} y_{q,j,i,C} &= x_{q,C} && \forall q \in W, j \in T, C \in A_q \\
z_i &\geq y_{q,j,i,C} && \forall q \in W, j \in T, i \in S_j \cup I_\emptyset, C \in A_q \\
\sum_{i \in I} z_i \cdot size(i) &\leq B
\end{aligned}$$

parameters:

$$W, A_{q \in W}, T, S_{j \in T}, B$$

variables:

$$x_{q,C} \in \{0,1\},\ y_{q,j,i,C} \in \{0,1\},\ z_i \in \{0,1\}$$

Figure 3.2: CoPhy binary integer program

CoPhy partly follows the general architecture of the index selection algorithms. It generates index candidates per query using well-known heuristics from the literature [35]. However, it creates configurations for each query separately. The cost of a query is estimated by calling the query optimizer. The binary program is then constructed and solved by a selected BIP solver. The modularity of CoPhy is a great architectural choice. In case a module becomes obsolete, it can be easily replaced by its superior version. CoPhy, along with other LP algorithms guarantees optimal solutions for any storage budget at the cost of exhaustively enumerating all relevant candidate combinations. Hence it is a good choice for systems with a small number of index candidates. When using CoPhy on large workloads, a strict candidate selection is required. Otherwise, the number of configurations can be too large to be handled by the BIP solver.

Example 1. Let's consider a simple database structure. As illustrated in Figure 3.3, we have tables `Company` and `Building`. The former table contains attributes `name`, `revenue` and `profit`, and the latter table consists of attributes `city`, `street` and `price`. Moreover, each one contains some basic information that is then queried.

The workload consists of three queries:

1. `SELECT * from Company WHERE profit <= 10 AND revenue > 20;`
2. `SELECT Name from Company WHERE revenue = 200;`

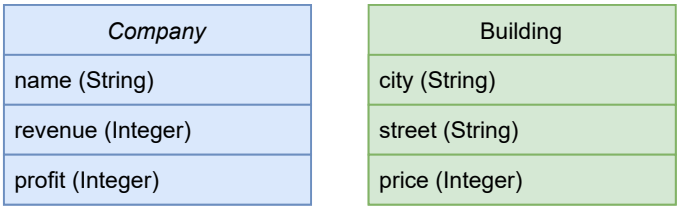

Figure 3.3: Example database structure

3. `SELECT count(*) from Company, Building where profit > price;`

Cophy's first module creates a set of index candidates for each query. The sets are based on the columns used in the queries. The exhaustive list of indexes that influence the query cost is used. { } represents a set of indexes, ( ) represents a single multi-column index, and [ ] represents an atomic configuration. In this particular case, the index candidates look like this:

- Candidates for q_1: { `(profit, revenue), profit, revenue` }
- Candidates for q_2: { `revenue` }
- Candidates for q_3: { `(profit, revenue), profit, price` }

Hence, the set of candidates is:

$$I := \{\texttt{(profit, revenue), profit, revenue, price}\}$$

And the workload is:

$$W := \{q_1, q_2, q_3\}$$

Cophy then constructs atomic configurations for each query:

$$A_1 := \{\texttt{[], [profit], [revenue], [(profit, revenue)]}\}$$
$$A_2 := \{\texttt{[], [revenue]}\}$$
$$A_3 := \{\texttt{[], [profit], [profit, price], [price], [(profit, revenue), price]}\}$$

We then compute costs for each candidate combinations (see Table 3.1 for A_1, Table 3.2 for A_2, and Table 3.3 for A_3) and the query it is used for. We used PostgreSQL and executed the queries on 100000 rows in each table. The costs are in milliseconds. Storage consumption are as listed in Table 3.4. The values are in kilobytes. Finally, our storage budget B is 3000.

We then construct the BIP according to the model illustrated in Figure 3.2, minimizing the execution cost while satisfying the budget constraint. If we set the same weights for each query, the optimal solution would be: `{price}`. Even though the `profit` is helpful in more queries, `price` lowers the cost the most in Q_3 which is the most expensive query. □

Table 3.1: Table of costs for A_1

Configuration	Cost
`[ ]`	35
`[profit]`	3
`[revenue]`	18
`[(profit, revenue)]`	2

Table 3.2: Table of costs for A_2

Configuration	Cost
`[ ]`	20
`[revenue]`	5

Table 3.3: Table of costs for A_3

Configuration	Cost
`[ ]`	> 100000
`[profit]`	1985
`[price]`	1648
`[profit, price]`	1232
`[(profit, revenue), price]`	1762

Table 3.4: Table of storage consumptions

Index	Storage consumption
`revenue`	920
`profit`	920
`(profit, revenue)`	2208
`price`	2100

3.2.2 Summary

There are multiple algorithms that solve the index selection problem using linear programming, namely CoPhy [9], Generalized Uncapacitated Facility Location Problem (GUFLP) [36], and Integer Linear Program from Ailamaki et al. (ILP) [37]. GUFLP, however, allows only a single index per query which makes it unsuitable for modern workloads. ILP, on the other hand, is similar to CoPhy but it creates substantially more variables which leads to a longer solving time. Thus, CoPhy is the best-performing algorithm in this category [9].

3.3 Greedy algorithms

In modern database systems, the number of potential index combinations can be prohibitively large, making it impossible to search the entire space of possible index selections. To address this challenge, researchers have proposed various greedy algorithms [30, 8, 7, 33]. These algorithms can produce results that are close to the optimal solution while running significantly faster than the optimal algorithms. Despite not being guaranteed to find the optimal solution, they provide an efficient way to approximate the best index selection. Hence they are

widely used in practice [7]. In this section, we explore the application of greedy algorithms, including their characteristics, different types, and their limitations.

3.3.1 Drop heuristics

Drop heuristics [30] is one of the oldest index selection algorithms. It does not guarantee the optimal solution but is fast and easy to implement. The main idea behind Drop heuristics (see Algorithm 1) is to consider all possible index candidates and then eliminate indexes that are not helpful for query processing. It works in a series of iterations, where in each iteration, it drops the index that contributes the least to the query processing performance. The main drawback of this algorithm is that it only considers single-column indexes. The original version drops indexes until there is no cost reduction in the workload execution time. However, this is not suitable for modern, heavy-read workloads. Therefore, a modified version of this algorithm has been introduced. It drops indexes until a certain size of the index set is reached [6].

Algorithm 1: Modern Drop Heuristics

Input: W – list of queries (workload)
I – list of all index candidates
k – number of indexes to be selected

```
1  RESULT := I
2  while length of RESULT > k do
3      lowest_cost := ∞
4      index_to_drop := None
5      for index in RESULT do
6          cost := calculate_cost(W, RESULT − {index})
7          if cost < lowest_cost then
8              lowest_cost := cost
9              index_to_drop := index
10     RESULT := RESULT − {index_to_drop}
11 return RESULT
```

Example 2. Let's consider the same database structure and workload as in Example 1. Once again, we have two tables, namely `Company` and `Building`, and queries q_1, q_2, and q_3.

Drop algorithm starts with the exhaustive set of all index candidates:

$$I = \{\texttt{profit, revenue, price}\}$$

We want to select 2 indexes, i.e., $k = 2$. The algorithm 1 starts with the candidate list I and iteratively drops the least helpful index until the desired number of indexes is reached. The first iteration considers the configurations listed in Table 3.5.

The configuration from Table 3.5 is then used by the *calculate_cost* function. Algorithm 1 then greedily drops the least helpful index. In this case, it is `revenue`. Hence the resulting index configuration is `{profit, price}`. The algorithm then reaches the required index count and stops. □

Table 3.5: Costs for configurations of size $|I| - 1$

Configuration	Q_1 **cost**	Q_2 **cost**	Q_3 **cost**	W **cost**
`{profit, revenue}`	3	5	1985	1993
`{profit, price}`	3	20	1232	1255
`{revenue, price}`	18	5	1648	1671

3.3.2 AutoAdmin

AutoAdmin algorithm [8] follows the general architecture of index selection algorithms. The main difference is that it starts with single-column indexes and then iteratively increases the index width until the required number of indexes is reached or the cost of the index set cannot be reduced any further. AutoAdmin architecture is shown in Figure 3.4. Although there are several versions of this algorithm [38, 7], the original version uses the number of indexes as the main constraint.

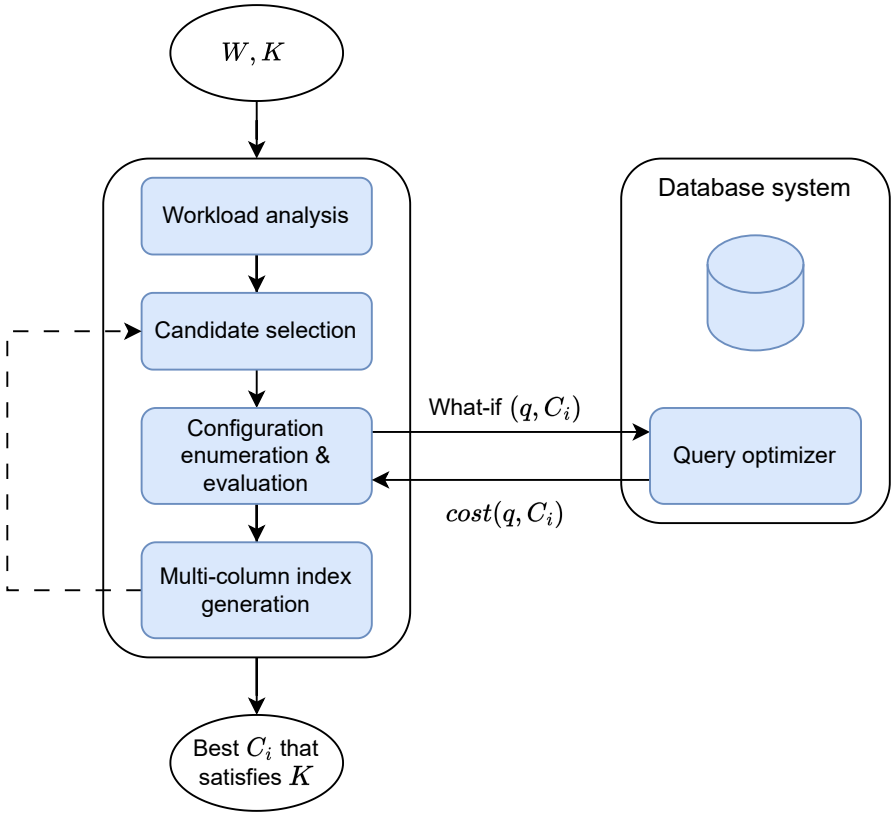

Figure 3.4: AutoAdmin architecture

The algorithm (see Algorithm 2) first determines index candidates for each query and then merges them into a single candidate set. The candidate selection is implemented in Algorithm 3. The algorithm considers potential candidates for each query, performs greedy enumeration and merges them into the final candidate list. The algorithm then greedily selects the best indexes from the candidate set until the required number of indexes is reached or the cost stops decreasing (see

Algorithm 2: AutoAdmin

Input: W – list of queries (workload)
max_index_count – number of indexes to be selected
max_naive – size of naive enumeration
max_index_width – maximum index width

```
1 P := {}
2 S := {}
3 for w ∈ [1, ..., max_index_width] do
4     I := select_index_candidates(W, P)
5     S := enumerate_configurations(W, I)
6     if w < max_index_width then
7         P := S ∪ create_multicolumn_indexes(W, S)
8 return S
```

Algorithm 3: AutoAdmin – function *select_index_candidates()*

Input: W – list of queries (workload)
P – list of potential candidates

```
1 function select_index_candidates(W, P):
2     S := {}
3     for q ∈ W do
4         // Select only those candidates from P that belong to q
5         P_q := filter_candidates_for_query(P, q)
6         S := S ∪ enumerate_configurations(W, P_q)
7     return S
```

Algorithm 4). The second iteration of the algorithm uses the index set from the previous iteration to build multi-column indexes of +1 width (see Algorithm 7). To reduce the number of new multi-column candidates, only indexes from the previous set can be the leading columns of the new multi-column indexes. In addition to approximating the query cost, AutoAdmin uses two interesting tricks to improve its performance. First, it uses a naive enumeration of index candidates for small configuration sizes (see Algorithm 5). Second, in the candidate selection phase, it uses the index selection algorithm itself to determine the best candidates for a single query(see Algorithm 3). It then performs a union of them.

Algorithm 4: AutoAdmin – function *enumerate_configurations()*

Input: W – list of queries (workload)
I – list of index candidates

```
1 function enumerate_configurations(W, I):
2     temp_indexes := enumerate_naive(W, I)
3     indexes := enumerate_greedy(W, temp_indexes, I − temp_indexes)
4     return indexes
```

Example 3. Let's have the same database structure and workload as in Example 1. We would like to select two indexes for this structure (i.e., $max\_index\_count = 2$), and set the other parameters to $max\_naive = 0$, $max\_index\_width = 2$.

Algorithm 5: AutoAdmin – function *enumerate_naive()*

```
Input: W – list of queries (workload)
       I – list of candidates that need to be enumerated
1  function enumerate_naive(W, I):
2      n := min(max_naive, |I|)
3      best_configuration := {}
4      best_cost := ∞
5      for i ∈ [1, ..., n] do
6          for C ∈ (I choose i) do
7              cost := calculate_cost(W, C)
8              if cost < best_cost then
9                  best_configuration := C
10                 best_cost := cost
11     return best_configuration
```

Algorithm 6: AutoAdmin – function *enumerate_greedy()*

```
Input: W – list of queries (workload)
       I – list of already enumerated candidates
       C – list of candidates that must be enumerated
1  function enumerate_greedy(W, I, C):
2      index_number := min(max_index_count, |I|)
3      best_index := None
4      best_cost := ∞
5      initial_cost := calculate_cost(W, I)
6      if length of I ≥ index_number then
7          return I
8      for index ∈ C do
9          cost := calculate_cost(W, I ∪ {index})
10         if cost < best_cost then
11             best_index := index
12             best_cost := cost
13     if best_cost < initial_cost then
14         C := C − {best_index}
15         I := I ∪ {best_index}
16         return enumerate_greedy(W, I, C)
17     return I
```

We start the algorithm by finding proper single-column index candidates. We do that for each query independently:

- Candidates for q_1: `{profit, revenue}`
- Candidates for q_2: `{profit}`
- Candidates for q_3: `{profit, price}`

Algorithm 7: AutoAdmin – function *create_multicolumn_indexes()*

```
Input: W – list of queries (workload)
       S – list of indexes on top of which we will build multi-column indexes
1  function create_multicolumn_indexes(W, S):
2      multicolumn_candidates := {}
3      for index ∈ S do
4          indexable_columns := get_indexable_columns(index.table())
5          indexable_columns := indexable_columns \ index.columns()
6          for column ∈ indexable_columns do
7              new_index := index ∪ {column}
8              multicolumn_candidates := multicolumn_candidates ∪
                {new_index}
9      return multicolumn_candidates
```

Next, we make a union of all candidates and get the set of candidates I which is `{profit, revenue, price}`. We then run the greedy algorithm to select the best indexes from I. We can use the cost values from the Tables 3.1,3.2, and 3.3, to compute the total cost of each index. Index `profit` achieves the cost of 2008, index `revenue` achieves the cost > 100000, and index `price` achieves the cost of 1703. AutoAdmin selects the index `price` as the best candidate. We then run the greedy algorithm again to select the second index. In this case, we select the index `profit`. The total cost of `{price, profit}` according to Table 3.5 is 1255.

The algorithm then generates two-column candidates. We have a single candidate `(profit, revenue)`. It does not lead to any greedy improvement, so by another set of iterations, we get the final solution `{price, profit}`. □

3.3.3 Anytime Database Tuning Advisor

Anytime Database Tuning Advisor (DTA) [7] is an extended version of AutoAdmin. DTA also selects index candidates for each query but it does it once including multi-column indexes. Then it considers indexes that might not be useful in the context of a single query but might be useful in the context of the workload. Then it greedily selects the most useful indexes. These indexes are then processed by a combination of multiple greedy algorithms. Indexes are ordered by their benefits so any time the process is stopped, the program returns the current best solution.

3.3.4 Extend algorithm

Extend [33] is one of the most recent greedy algorithms for index selection. It criticizes the premature candidate pruning that occurs in the candidate selection step of the general architecture (Figure 3.1). Although the pruning makes the algorithms faster, it usually leads to suboptimal selections, since the pruned candidates may appear useful. Therefore, the Extend algorithm does not follow the general architecture but uses a step-wise constructive approach. The pruning of the candidates occurs as late as possible in order to maximize the chances of finding the optimal solution. This approach allows the Extend algorithm to

consider a larger pool of candidates and make better-informed decisions during the selection process.

In each step of the algorithm, Extend considers adding a single-column index to the current set of indexes S or extending one of the existing indexes in S with a new attribute. It does so by calculating the ratio of the execution cost and the memory consumption of the new index set. The original paper [33] also considers the reconfiguration cost of the new index set, which is denoted as $R(S^*, S)$. In this context, S^* is the new index set, and S is the current index set. The reconfiguration cost is the cost of changing the current index set to the new one. The final ratio for an index i is described by Equation 3.1.

$$\frac{R(S, \emptyset) + \sum_{q \in W} cost(q, S) - \sum_{q \in W} cost(q, S \cup \{i\}) - R(S \cup \{i\}, \emptyset)}{size(S \cup \{i\}) - size(S)} \tag{3.1}$$

The index that has the best ratio is added to the current set of indexes S. The complete algorithm is shown here:

1. Start with an empty set S.
2. Find an index from the set of single-column candidates I that minimizes the ratio of its cost and memory consumption. Add the index to the set S.
3. For each column in I and each element from S, consider the following options:
   (a) Add the index to the set S.
   (b) Append the selected attribute from I to an element from S to form a multi-column index.

   This will result in a new set S^*.
4. From all S^* sets created in step 3, select the set that maximizes the Ratio 3.1. This set becomes the new S.
5. Repeat steps 3 and 4 until the budget is reached or no improvement can be made.

This algorithm can also be used for online index selection. However, being focused on offline index selection, we do not consider reconfiguration cost.

Example 4. We will once again use the example from Example 1. Let our storage budget B be 3000.

Table 3.6: Table of single-column index candidates

Index candidate	**Total cost**	**Storage size**	**Benefit ratio**
`profit`	2008	920	~ 107
`revenue`	> 100000	920	~ 0
`price`	1703	2100	~ 46

The algorithm first compares different benefit ratios of single-column index candidates (see Table 3.6). The first selected index is `profit`. Other algorithms would pick `price` because it minimizes the total cost. However, Extend prefers indexes with a higher benefit per storage unit.

The current cost of S is 2008 and the current storage size is 920. So we move to step 3, construct new index configurations, and compute the benefit ratios relative to the current S (see Table 3.7).

Table 3.7: Benefit ratios of extended index candidate sets

Index candidate	Total cost	Storage size	Benefit ratio
`{profit, revenue}`	1993	1840	0.02
`{profit, price}`	1255	3020	0.35
`{(profit, revenue)}`	2208	2022	-0.01

Although `{profit, price}` has the highest relative benefit ratio, it exceeds the storage budget. Therefore, we select `{profit, revenue}` as the next index. This leads to a suboptimal solution, since `{price}` would have been a better choice. □

3.4 Reinforcement Learning based algorithms

Reinforcement Learning (RL) is a branch of machine learning that focuses on how an agent can learn to maximize its reward in a given environment. In RL, an agent interacts with the environment and receives rewards or punishments based on the actions it takes. In the index selection problem, we use query costs to measure the performance of the system and sequentially improve the performance by selecting new combinations of indexes. RL models can use these costs to optimize decisions and adjust their behavior accordingly, making them well-suited for the index selection problem.

3.4.1 NoDBA

Researchers from Saarland Informatics Campus introduce a deep reinforcement learning (DRL) algorithm called NoDBA [10] that learns to select indexes for a given workload. On a high level, it simulates the work of database administrators (DBAs) who change the parameters of the database based on the rewards or punishments they receive from the system. The main components of the DRL algorithm are:

- **Input to the neural network:** workload and the current index configuration
- **Actions that agent can take:** create a secondary index
- **Reward function:** total workload run-time improvement after adding the index
- **Hyper parameters:** the number of hidden layers, neurons, iterations, etc.

The essential part of the algorithm is the learning process (see Figure 3.5) which uses the main ideas of RL. It is an iterative process where each step i is described as follows:

1. Based on the workload and the current index configuration C_i, the neural network predicts the next action a.
2. The predicted action a is applied to the current index configuration. A new configuration C_{i+1} is created.
3. Reward r is computed using the reward function f on the new configuration C_{i+i}.
4. The neural network is trained using C_i, C_{i+1}, a and the reward r. The program then returns to step 1.

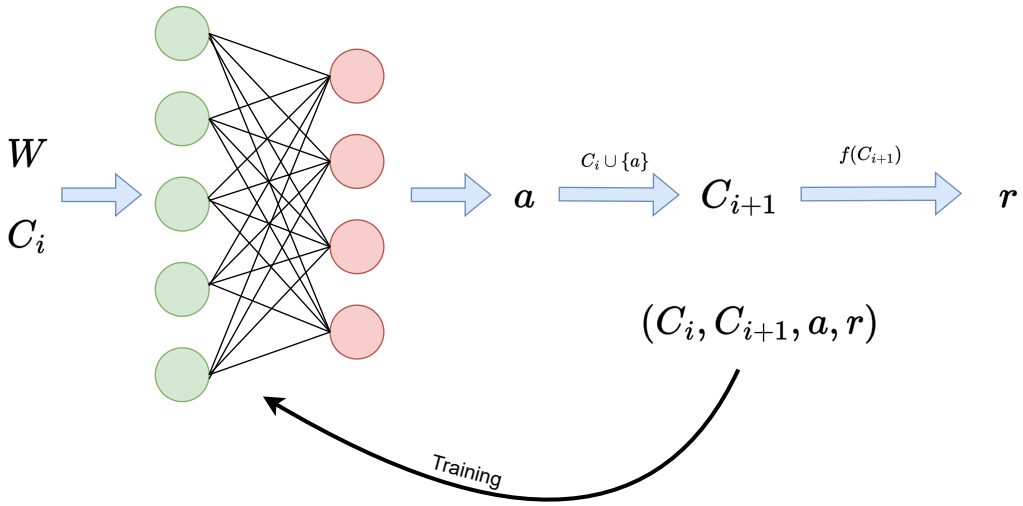

Figure 3.5: NoDBA learning process

Input to the network is represented as a matrix of size $n \times m$ where n is the number of queries and m is the number of columns in the database. Each cell then describes the selectivity of a column based on the current query. It is the ratio of selected rows based on the column predicate versus the total number of rows in the table where the column is located. If the query does not select a specific column, the value is 1. The current index configuration is also included in the network input and encoded as a bit list of size m. As could be already inferred, the algorithm only considers single-column indexes. Although it is possible to encode multi-column indexes, the combination of the current design and the increased index width leads to extremely high learning times [11].

Limited experiments show that the algorithm performs well on highly selective workloads [10]. It however does not scale well to bigger workloads and has many limitations. For example, no support for multi-column indexes and long training times. This was however the first paper to use RL for index selection, so it is a good starting point for further research.

3.4.2 Budget-aware RL algorithm

Despite the great complexity of ISP algorithms, the majority of their run-time is spent on what-if calls. Experiments show that what-if calls take between 75% and 93% of the total execution time of modern ISP algorithms [39]. Hence, there has been a recent interest in budget-aware algorithms that can limit the number of database optimizer calls. A novel paper uses reinforcement learning to select indexes while satisfying the budget [11]. It works with the general architecture 3.1 of the cost-based index selection algorithm and focuses on rebuilding the candidate enumeration component. Therefore it assumes that index candidates are already selected, and it only needs to select the best combination of indexes.

In order to lower the number of what-if calls, it uses an upper bound approximation of the actual query costs. The algorithm assumes that the index configuration costs are monotone, thus the approximation is computed as the minimum cost over all subset configurations of the current configuration. This approximation is called *derived* cost:

$$derived(q, X) = \min_{X' \subseteq X} cost(q, X')$$

The main goal of the algorithm is to decide what configurations should be approximated and what configuration costs should be calculated by the database optimizer. The information about the known what-if costs is stored in the *Budget allocation matrix*. It is a $n \times m$ matrix where n is the number of queries and m is $2^{|I|} - 1$ where I is the set of index candidates. Cell with a value of 1 means that the cost of the query q with the configuration X is known and 0 means that the cost must be approximated.

To set values in the matrix the algorithm must find a balance between allocating the budget to configurations that contain the best indexes so far and configurations that do not contain the best indexes but contain unexplored indexes with similar costs. This problem is known as the exploration vs exploitation dilemma and is solved by formulating it as MDP. The algorithm then uses MCTS to find actions that maximize the expected reward.

MDP formulation is quite straightforward. States $\mathcal{S}$ are defined as all index configurations in the search space. Precisely, it is a set of all subsets of the index candidates I. Actions $\mathcal{A}(s)$ for a state $s \in \mathcal{S}$, are defined by the missing indexes in the current state. Hence, from a single state whose configuration size is k, we can get to all index candidates subsets of size $k + 1$. As the transitions are deterministic, transition probabilities $\mathcal{P}$ of the actions are set in this manner: $Pr(\texttt{from state } s \texttt{ by action } a \texttt{ to state } s \cup \{a\}) = 1$. The expected return of a state s is the expected cumulative reward when starting from s and taking actions according to some policy. The algorithm computes the expected return as the expected percentage improvement of all configurations that contain the indexes of the state s as a subset. The percentage improvement of a state s represented by the configuration X is computed as:

$$\left(1 - \frac{\sum_{q \in W} derived(q, X)}{\sum_{q \in W} derived(q, \emptyset)}\right) \times 100$$

The goal of RL is to find an optimal policy that maximizes the expected return when starting from the initial state $\emptyset$.

MCTS is then used to find the optimal action selection strategy. It is a tree search algorithm that uses a Monte Carlo method to traverse the tree. A part of the MCTS is based on a random sampling of the index candidates. The authors' implementation chooses actions randomly but assigns higher probability to actions that have a lower average cost. To optimize the number of what-if calls even more, the cost for a configuration X picked in an episode is approximated by derived costs, and a what-if cost is called for only a single query q. The probability of selecting a specific query is proportional to its derived cost. The final cost is then:

$$cost(q, X) + \sum_{q' \in W \setminus \{q\}} derived(q', X)$$

3.5 Comparison of the algorithms

All of the algorithms presented in this chapter were originally developed and tested on the relational model. Our goal in this section is to list the key features relevant to multi-model databases and determine which algorithms are suitable for other data models.

Table 3.8 illustrates the comparison of the algorithms and their properties that are relevant for multi-model databases. As we can see, most of the researched algorithms belong to the greedy family since the index selection problem is NP-complete, and searching for the optimal solution is not feasible [31]. One of the crucial properties of the described algorithms is whether they follow the general architecture 3.1. We can see a pattern across the table that the algorithms that follow the general architecture are more flexible and can be easily adapted to other data models.

Each of the algorithms we compared supports single-column indexes, which is also applicable to other data models since all of them support the concept of an indexable attribute. However, the algorithms such as Drop and NoDBA do not support multi-column indexes. This causes a loss of performance in relational models and makes them less suitable for other data models that support multi-attribute indexes. None of the algorithms support partitioning keys. It is understandable since the authors developed them solely for single-node relational databases. However, it poses a problem for distributed DBMSs such as, e.g., document and columnar stores.

Moreover, the algorithms do not consider multiple index types, a crucial feature of many DBMSs. We can solve this problem, e.g., in the candidate selection phase by selecting multiple index types for a single attribute and then enumerating the candidates.

Another important feature of the researched algorithms is how they select the candidates. Some older proposals, such as AutoAdmin and CoPhy, select candidates based on individual queries, whereas the newer algorithms, such as NoDBA and DTA, select candidates based on the workload. Both approaches have their advantages and disadvantages. The workload-based approach is more complex but can find beneficial indexes for the whole workload. On the contrary, the query-based approach is more straightforward but can miss indexes that are not beneficial for individual queries but are beneficial for the whole workload.

Some of the non-essential traits are the support of anytime property and the

Table 3.8: Comparison of index selection algorithms

	CoPhy [9]	**Drop** [30]	**AutoAdmin** [8]	**DTA** [7]	**Extend** [33]	**NoDBA** [10]	**MCTS** [11]
Approach type	ILP	Greedy	Greedy	Greedy	Greedy	ML	ML
Follows architecture 3.1	Yes	No	Yes	Yes	No	No	Yes
Main constraint	Storage	Indexes	Indexes	Many[1]	Storage	Indexes	What-if calls
Single-column indexes	Each algorithm supports single-column indexes						
Multi-column indexes	Yes	No	Yes	Yes	Yes	No	Yes
Index types	B-Tree, Hash, B+ Tree, Bitmap, Inverted index						
Best index type	No	No	No	No	No	No	No
Partitioning keys	No	No	No	No	No	No	No
Optimal solution	Yes[2]	No	No	No	No	No	No
Stop any time	Yes	No	Yes	Yes	Yes	No	No
Candidates context	Query	Workload	Query	Workload	Workload[5]	Workload	Depends
Online selection	No	No	No	No	Yes	Yes	No
Relational DBMSs	Yes	Partial	Yes	Yes	Yes	Partial	Yes
Graph DBMSs	Yes[3]	Partial	Yes[4]	Yes[4]	Partial	Partial	Yes[4]
Key/Value DBMSs	None of the algorithms supports manual construction of secondary indexes						
Document DBMSs	Yes[3]	Partial	Yes	Yes	Yes[3]	Partial	Yes
Columnar DBMSs	Each algorithm supports single-column indexes						

[1] DTA can be configured to use many constraints at the same time. It supports indexes, storage, and running time.
[2] CoPhy does find the optimal solution based on the candidates that were provided. Usually, the number of candidates is limited.
[3] Generally supports the data model but is not suitable for it.
[4] Generally supports the data model, however, at least three components of the algorithm need to be adjusted.
[5] Extend does not contain a candidate selection phase. However, it considers potential indexes in the context of the workload.

support of online selection. The anytime property ensures that the algorithm can stop at any time and still produce a valid solution. This is quite useful for NoSQL and multi-model DBMSs since they may work with large datasets, and the index selection process can take a long time. The online selection feature allows the algorithm to select indexes on the system where the workload and the database architecture constantly change. The algorithm must then consider the changes and drop or add indexes depending on the current workload. From the algorithms we discussed, only NoDBA and Extend support online selection.

Index selection in Relational DBMSs As all of the researched algorithms were developed for relational DBMSs, they are highly optimized for this specific model. Drop and NoDBA, however, have some limitations when considering multi-column indexes. Otherwise, there are no main downsides to using these algorithms for the relational model. We must consider minor alterations when implementing the algorithms with specific database systems, such as PostgreSQL, Oracle, or MySQL.

Index selection in Graph DBMSs Graph model also groups data into independent units, but in this case, the units are represented as nodes instead of rows in a table. We can then easily add edges, allowing for more flexible and expressive modeling of complex relationships.

We can still use the researched algorithms for graph models. However, we need to make some adjustments based on the general architecture from Figure 3.1. First, we need to change the workload analysis phase. Graph databases, such as Neo4j, use a query language called Cypher. It is a declarative language that allows us to express complex graph traversals in a single query. This is a significant difference from the relational model, where we need to use multiple queries to express complex relationships. Hence the query analysis component must be adjusted to support the parsing of node and edge property predicates, such as the following example:

```
MATCH
    (t:Actor {name:"Tom"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(p:Actor)
WHERE
    p.gender = "male" AND p.born <= 1985
RETURN
    DISTINCT p.name
```

Second, we need to adjust the candidate selection phase. The scenario is similar to the document DBMSs, where we need to add new index types and take into account their limitations in order to avoid generating invalid candidates. Lastly, the candidate enumeration does not need to be changed, except for the algorithms that do not follow the general architecture, such as Extend, Drop, and NoDBA. The crucial change that influences each one of the researched algorithms is cost estimation. The cost estimation in graph DBMSs is more complicated than in other systems since the queries can be based on graph traversals and complex relationships between nodes. However, Neo4j provides a sufficient API for cost estimation [40]. The query plan does not provide the expected execution plan

but provides all operations that the query will perform. After the aforementioned adjustments, we can use any of the algorithms for graph models. As the number of candidate indexes might be large, CoPhy has to implement a more strict candidate selection phase. That will however lead to suboptimal solutions, making CoPhy less suitable for graph models.

Index selection in Key-Value DBMSs Key-value stores are the simplest of all the database systems we discuss. Therefore, there is little need for secondary indexes, since queries are usually based on the primary key. However, some key-value databases, such as Redis, support more complex data structures, which might require secondary indexes. Secondary indexes can be manually implemented by suitable data structures (hash tables, sorted sets). Hence, the index selection problem is replaced by the problem of choosing the right data structures and altering the queries to fit them. Although we might be able to find a bijection between the index selection problem and the data structure selection problem, the researched algorithms are not originally designed for this problem. Thus, it is out of the scope of this thesis to create a new algorithm that would construct the data structures and alter the queries.

Index selection in Document DBMSs Workloads in document DBMSs do not use relations among collections, making the queries generally simpler than in relational DBMSs. On the other hand, document DBMSs use new kinds of indexes, such as inverted indexes and geospatial indexes, which are not explicitly supported by any of the algorithms. Hence, when using the algorithms for document DBMSs, we must make them aware of these new index types. We can do that, for instance, by changing the candidate selection phase to include the new types. Additionally, index types such as hash and text indexes do not support multi-attribute indexes. Hence we must modify Extend algorithm to skip these indexes when extending the existing candidates. Drop and NoDBA only support single-field indexes, making them impractical for document stores. AutoAdmin and DTA support multi-attribute indexes and follow the general architecture, making them the most suitable for document stores. CoPhy can also be applied to the document DBMSs, but the increased number of index types makes the candidate enumeration phase more expensive. With proper candidate selection, we can also use MCTS for document stores. It might be a good choice since the what-if calls are expensive in document models, and MCTS can reduce their number. ML-based algorithms also show great potential for document models since there are a lot of index types and they can learn strategies to avoid unsuitable ones.

Index selection in Columnar DBMSs Distributed column-based systems use distributed indexes that store a partitioning key along with the indexed value. The key is used to partition the data across the cluster. This poses a challenge to index selection algorithms because they do not support data partitioning. Many modern distributed columnar databases solve this problem by partitioning the data based on the primary key. Secondary indexes are then co-located with the original data across the cluster [23]. Thus, the index selection algorithms do not have to consider data partitioning. However, one problem that the algorithms

need to address is that they can no longer rely on cost estimates because the distributed nature of the system cannot guarantee the same performance for every execution of the same query. Queries in the columnar model are much simpler than in the relational model, which makes the index selection problem easier. Additionally, Cassandra does not support multi-column indexes. Therefore algorithms such as DTA, MCTS, and NoDBA may even be deemed excessive or disproportionate for this kind of database systems. Drop and CoPhy, on the other hand, are a great fit for the columnar DBMSs. Drop only considers single-column indexes, and CoPhy can quickly find the optimal solution for the relatively small number of index candidates.

4. Open questions and challenges

Our analysis of the state-of-the-art index selection algorithms and their applicability to different data models revealed several open questions and challenges. In this chapter, we discuss these questions and challenges and propose possible directions for future research. We group the questions and challenges into five categories: *general*, *workload parsing*, *candidate selection*, *query cost*, and *candidate enumeration*. The last four categories are based on the key components of the general architecture of ISP algorithms (Figure 3.1).

- *General questions and challenges*
  - With the rise of big data, the iterative methods for index selection are becoming increasingly impractical due to their high computational cost and time-consuming nature. We discovered that CoPhy [9] is unsuitable for most of the researched DBMSs. Its time complexity rapidly rises with the number of index types. DRL algorithms, such as NoDBA [10], although just partially successful, are a promising direction for future research.
  - In columnar and key-value DBMSs, such as Cassandra and Redis, designing an appropriate *schema* rather than using secondary indexes is more important [23, 16]. Hence, multi-model ISP algorithms should be able to recommend a schema for the database. This is a challenging task because schema design is a complex process that requires a deep understanding of the data model and the workload. Another solution might be to find a bijection between the recommended secondary index and the schema. In Cassandra, for example, a multi-column index can be interpreted as a composite primary key.
- *Workload parsing*
  - Many DBMSs use different query languages such as Cypher, Cassandra Query Language (CQL), and MongoDB query language. We need to be able to parse the queries and extract index candidates from them. This can be solved, for instance, by using a specific query parser for each query language and then passing the potential index candidates to the candidate selection component.
- *Candidate selection*
  - The algorithms we researched officially do not differentiate between various index types. This poses a challenge for DBMSs that support a wide variety of index types, and their workload can significantly benefit from using them. Algorithms should be able to find proper index types for each candidate. We can solve this problem, e.g., by generating a set of candidates for each attribute according to the available index types. It will, however, increase the number of candidates, thus increasing the computational cost of the algorithm as well. Therefore, ML algorithms might be a good fit for these DBMSs as they can learn the strategies for preferring certain index types over others.

- *Query cost*
  - Query cost computation is the main challenge of the index selection problem. It requires the indexes to be built and the query to be executed. This is a time-consuming process, and executing it for every index configuration is not feasible. The current state-of-the-art algorithms speed up the process by approximating the query cost using the what-if calls and virtual indexes. These tools are available in relational DBMSs, but their availability in other database systems is limited. To use these algorithms in other DBMSs, we first need to research the possible ways of estimating the query cost for each model.
  - Even if we had tools to estimate the query cost, they would still be slow and not accurate enough. Hence, algorithms that are optimized for minimizing the number of what-if calls, such as MCTS [11], would be a great choice. Additionally, we can try to use a novel approach that uses machine learning to predict the query cost [41]. This might be the best solution, as the modern DBMSs have robust schemas, so the learning time can easily match the time needed to execute all the what-if calls.
  - All of the algorithms we researched assume that the workload is read-heavy, and they do not optimize for writing time. Some of them, such as Extend consider reconfiguration cost [33]. As columnar DBMSs are write heavy, we might need to consider write cost as well and include the insertion queries and their weights in the workload.
- *Candidate enumeration*
  - The enumeration is the part that is the easiest to be generalized for all the models. It just receives a set of candidates and their costs and returns the optimal configuration. The only challenge is finding the most optimal enumeration algorithm for each model. Some models might benefit from a greedy approach that selects the candidate with the absolute highest cost reduction, while others might benefit from a more sophisticated approach such as the benefit ratio presented in Extend [33]. Additionally, as the enumeration component closely works with the query cost, we need to minimize the number of what-if calls. Therefore, MCTS [11] might be a good choice for this component as well.

Conclusion

In this thesis, we made a step further towards the development of efficient and possibly autonomous multi-model DBMSs, as envisioned in [1]. We did so by analyzing the state-of-the-art index selection algorithms which are currently used in relational DBMSs. We then considered their applicability to other database systems, namely graph, key-value, document, and columnar DBMSs. Thanks to the analysis, we were able to formulate open questions, challenges, and ideas that might help in the future development of index selection algorithms for multi-model database systems. We conclude our work by summarizing the main contributions of this thesis and outlining future work.

- *Analysis of indexing in various DBMSs* We thoroughly described five most common data models and selected one representative DBMS for each of them. We then analyzed their indexing strategies. The analysis revealed that the indexing strategies differ in the implementation details and supported index types.
- *Analysis of existing approaches* We analyzed seven pioneering index selection algorithms and grouped them into three categories based on their approach: linear programming, greedy, and machine learning. We then formulated a general architecture of an index selection algorithm that generalizes the key components of the analyzed algorithms. Moreover, we provided a comprehensive explanation of how each algorithm works and what are its advantages and disadvantages.
- *Verification of their broader applicability* We statically analyzed whether the algorithms can be used for other DBMSs and what measures would be needed to make them applicable.
- *Definition of open questions* Our research revealed that several algorithms might be suitable for multiple data models. However, some challenges and open questions still need to be solved to develop an efficient index selection algorithm for multi-model DBMSs.

Future work

In our future work, we plan to solve the open questions that we formulated in chapter 4. Our main focus will be on the challenges that are closely related to the general architecture of ISP algorithms, since they affect the key components of the researched algorithms. Once we solve these challenges, we will be able to implement the algorithms and test their performance on various DBMSs. We will then iteratively refine a single algorithm that performs well on multi-model database systems.

Bibliography

[1] Irena Holubova, Pavel Koupil, and Jiaheng Lu. Self-adapting design and maintenance of multi-model databases. In *Proceedings of the 26th International Database Engineered Applications Symposium*, IDEAS '22, page 9–15, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Yu Yan, Hongzhi Wang, Yutong Wang, Zhixin Qi, Jian Ma, Chang Liu, Meng Gao, Hao Yan, Haoran Zhang, and Ziming Shen. Multi-sql: An automatic multi-model data management system. In Bohan Li, Lin Yue, Chuanqi Tao, Xuming Han, Diego Calvanese, and Toshiyuki Amagasa, editors, *Web and Big Data*, pages 451–455, Cham, 2023. Springer Nature Switzerland.

[3] Pavel Koupil and Irena Holubová. A unified representation and transformation of multi-model data using category theory. *J. Big Data*, 9(1):61, 2022.

[4] Pavel Koupil, Sebastián Hricko, and Irena Holubová. A universal approach for multi-model schema inference. *J. Big Data*, 9(1):97, 2022.

[5] Guoliang Li, Xuanhe Zhou, and Lei Cao. Ai meets database: Ai4db and db4ai. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2859–2866, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proc. VLDB Endow.*, 13(12):2382–2395, jul 2020.

[7] Surajit Chaudhuri and Vivek Narasayya. Anytime algorithm of database tuning advisor for microsoft sql server, June 2020.

[8] 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, page 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[9] A. Ailamaki D. Dash, N. Polyzotis. Cophy: A scalable, portable, and interactive index advisor for large workloads. *Proceedings of the VLDB Endowment*, 4:362–372, 2011.

[10] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. The case for automatic database administration using deep reinforcement learning, 2018.

[11] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. Budget-aware index tuning with reinforcement learning. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1528–1541, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Db-engines ranking - popularity ranking of database management systems. https://db-engines.com/en/ranking. Accessed: 2023-04-10.

[13] Jaroslav Pokorný and Ivan Halaška. *Databázové systémy*. ČVUT, 1998.

[14] PostgreSQL Development Team. Postgresql: Documentation: 14: Index types. https://www.postgresql.org/docs/current/indexes-types.html, 2021. Accessed: May 3, 2023.

[15] Neo4j. Query tuning with indexes. https://neo4j.com/docs/cypher-manual/current/query-tuning/indexes/, 2021. Accessed: 2023-04-25.

[16] Redis. Indexes. https://redis.io/docs/manual/patterns/indexes/, 2021. Accessed: 2023-04-30.

[17] Ecma International. JavaScript Object Notation (JSON), 2013. http://www.JSON.org/.

[18] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008.

[19] Pavel Contos and Martin Svoboda. JSON schema inference approaches. In Georg Grossmann and Sudha Ram, editors, *Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3-6, 2020, Proceedings*, volume 12584 of *Lecture Notes in Computer Science*, pages 173–183. Springer, 2020.

[20] MongoDB. Mongodb manual: Indexes. https://www.postgresql.org/docs/current/indexes-types.html, 2021. Accessed: April 28, 2023.

[21] Couchbase. Indexes. https://docs.couchbase.com/server/current/learn/services-and-indexes/indexes/indexing-and-query-perf.html, 2023. Accessed: May 2, 2023.

[22] MongoDB. Mongodb sharding. https://www.mongodb.com/docs/manual/sharding/. Accessed: 2023-04-15.

[23] Duy Hai Doan. Cassandra native secondary index deep dive. https://www.datastax.com/blog/cassandra-native-secondary-index-deep-dive, 2016. Accessed: April 22, 2023.

[24] Cassandra. Cassandra documentation. https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html, 2022. Accessed: April 22, 2023.

[25] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, jun 1996.

[26] Apache Cassandra. Secondary indexes. https://cassandra.apache.org/doc/latest/cassandra/cql/indexes.html, 2021. Accessed: April 24, 2023.

[27] DataStax. Using multiple secondary indexes. https://docs.datastax.com/en/archived/cql/3.3/cql/cql_using/useMultIndexes.html, 2015. Accessed: April 24, 2023.

[28] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[29] Jiaheng Lu and Irena Holubová. Multi-model databases: A new journey to handle the variety of data. *ACM Comput. Surv.*, 52(3), jun 2019.

[30] Kyu-Young Whang. Index selection in relational databases. In *Proc. Int'l Conf. on Foundations of Data Organization*, pages 369–378, Kyoto, Japan, May 21-24 1985.

[31] Gregory Piatetsky-Shapiro. The optimal selection of secondary indices is np-complete. *SIGMOD Rec.*, 13(2):72–75, jan 1983.

[32] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. Index interactions in physical design tuning: Modeling, analysis, and applications. *Proc. VLDB Endow.*, 2(1):1234–1245, aug 2009.

[33] Rainer Schlosser, Jan Kossmann, and Martin Boissier. Efficient scalable multi-attribute index selection using recursive strategies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1238–1249, 2019.

[34] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: a hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1313–1323, 2004.

[35] Nicolas Bruno, César A. Galindo-Legaria, and Milind Joshi. Polynomial heuristics for query optimization. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 589–600. IEEE Computer Society, 2010.

[36] Alberto Caprara and Juan José Salazar González. A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem. *Top*, 4:135–163, 06 1996.

[37] Anastasia Ailamaki and Stratos Papadomanolakis. An integer linear programming approach to database design. *Proceedings of the 2nd International Workshop on Self-Managing Database Systems (SMDB 2007)*, 2007.

[38] Sanjay Agrawal, Nico Bruno, Surajit Chaudhuri, and Vivek Narasayya. Autoadmin: Self-tuning database systems technology. In *Data Engineering Bulletin*. IEEE Computer Society, January 2006.

[39] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. Efficient use of the query optimizer for automated physical design. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1093–1104. VLDB Endowment, 2007.

[40] Neo4j. Query execution plan. https://neo4j.com/docs/cypher-manual/current/execution-plans/, 2021. Accessed: 2023-04-25.

[41] Jiachen Shi, Gao Cong, and Xiaoli li. Learned index benefits: Machine learning based index performance estimation. *Proceedings of the VLDB Endowment*, 15:3950–3962, 01 2023.

List of Figures

1.1 Possible index properties in Neo4j 6

3.1 General architecture of the index selection algorithms 14
3.2 CoPhy binary integer program 16
3.3 Example database structure 17
3.4 AutoAdmin architecture 20
3.5 NoDBA learning process 26

List of Tables

1.1 Comparison of different DBMSs 10

3.1 Table of costs for A_1 18
3.2 Table of costs for A_2 18
3.3 Table of costs for A_3 18
3.4 Table of storage consumptions 18
3.5 Costs for configurations of size $|I| - 1$ 20
3.6 Table of single-column index candidates 24
3.7 Benefit ratios of extended index candidate sets 25
3.8 Comparison of index selection algorithms 29

List of Abbreviations

ACID Atomicity, Consistency, Isolation, Durability. 4

API Application Programming Interface. 12

BIP Binary Integer Program. 15–17

BRIN Block Range Index. 10

CQL Cassandra Query Language. 33

DBA Database Administrator. 25

DBMS Database Management System. iii, 2–11, 28–35

DRL Deep Reinforcement Learning. 25, 33

DTA Anytime Database Tuning Advisor. 23, 28, 29, 31, 32

GUFLP Generalized Uncapacitated Facility Location Problem. 18

ILP Integer Linear Program. 18, 29

IoT Internet of Things. 7

ISP Index Selection Problem. 12, 14, 27, 33, 35

JSON JavaScript Object Notation. 7

LP Linear Program. 15, 16

LSMT Log-Structured Merge Tree. 8, 10

MCTS Monte Carlo Tree Search. 27–29, 31, 32, 34

MDP Markov Decision Process. 27

ML Machine Learning. 29, 31, 33

MMDBMS Multi-Model Database Management System. 3, 9

NoSQL Not only SQL. 2, 4, 5, 9, 30

RL Reinforcement Learning. 25–27

XML eXtensible Markup Language. 7