# Analysis of Database Algorithms for Performance and Access Optimization

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

**Andrei Stan**
Spring, 2021

**Technical Project Team Members**

**Susan Le**
**Bradley Lund**

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signature ______________________________________________ Date __________

**Andrei Stan**

Approved ______________________________________________ Date __________

**Nada Basit, Department of Computer Science**

# Analysis of Database Algorithms for Performance and Access Optimization

**Le, Susan**
Computer Science
University of Virginia
Charlottesville, VA
sl3ub@virginia.edu

**Lund, Bradley**
Computer Science
University of Virginia
Charlottesville, VA
wbl2yj@virginia.edu

**Stan, Andrei**
Computer Science
University of Virginia
Charlottesville, VA
as4rn@virginia.edu

## ABSTRACT

Database design is optimized in several areas for different contexts. These areas include but are not limited to indexing and querying. Algorithmic decisions can optimize several of these areas but with trade-offs in others. To identify these trade-offs, we will investigate and categorize algorithms, such as trees, to determine which are most and least suited for each optimization. This will be done using a document analysis of relevant literature. Algorithmic efficiency will be evaluated by metrics such as time and space complexity and feasibility of implementation. Consolidating information about existing algorithmic efficacy will allow us to reveal the areas with most significant potential for improvement. We will describe how these improvements may benefit respective fields, including big data and distributed systems.

## INTRODUCTION

Big data is based around processing large quantities from a database. How exactly do the algorithms involved with big data retrieve data quickly, efficiently, and accurately from the database? In this paper, we will investigate several algorithmic implementations for querying and heuristics for making this process quicker and more reliable. We will first investigate indexing, a core component of fast data retrieval. Then, we will connect this to the querying process itself, by which relevant indexed data is extracted from the database. Finally, we will explore optimizations in the context of big data and distributed systems to determine practical and feasible applications of these techniques.

## 1 INDEXING

Indexing decisions are a vital part of database design. Efficient indexing ensures quick access to information that users of the database most frequently want to access. Indexing is usually performed using trees, especially B+ trees. Many variants of these trees exist to fulfill specific niches, such as processing WGIs and graphs. B-Trees can also be augmented with logging capabilities to improve recovery after failure.

### 1.1 VR-Tree: A novel tree-based approach for modeling Web Query Interfaces[6]

A variant called "Visual Reduced Trees" (VR-Trees) can be used for modeling Web Query Interfaces (WQIs). WQIs serve as a link between HTML form input and results retrieved from the database. By understanding the intent and semantics of each individual WQI, a unified WQI can be created representing the entire domain of data access instead of one specific type of query, a task referred to as "WQI modeling." Modeling consists of two subtasks: parsing (identifying visual components) and components modeling (mapping these components to a relevant data structure). VR-Trees can automate these tasks; to automate VR-Tree creation, an algorithm can be applied to data structures already present in the rendering engine of web browsers.

The VR-Tree creation algorithm starts with a pre-processing phase that prunes a browser's render tree to only relevant information nodes (RBlock, RTable, RTableCell, and RLine) and data nodes (RWord and RUIControl). To construct the VR-Tree, six heuristic rules are applied to this reduced render tree:

1. WQIs are organized top-down and left-to-right, with depth-first search used for organization.
2. WQI fields are leaves and may or may not have an associated label, identified as consecutive RWord nodes.
3. The label associated with a radio button or check box is to the right of that field.
4. The label associated with a text-input field is almost always to the left or above that field.
5. A type selection list label is usually to the left.
6. A group label is usually above or left of the group.

To create the VR-Trees, several functions are defined. AssignLabelRBlock and AssignLabelRTable both use recursion to match their respective types (RBlock and RTableCell) with labels based on the heuristics. From these, FindGroupsInBlock and FindSuperGroups are used to group similar field/label nodes and then further hierarchically classify these groups.

From these functions, two algorithms are created. The first identifies fields/labels, while the second identifies

```
Algorithm 1 BuildPartialVR-Tree( RNode node)
GroupNode  $g_n \leftarrow$ null;
 if node instanceof RBlock
   then
     if node contains RLine nodes
       then {$g_n \leftarrow$ AssignLabelRBlock($node.childrenList$)};
     else
       $g_n \leftarrow new$ GroupNode
       for each $n_i \in node.childrenList$
         do
           GroupNode $gn_i \leftarrow$ BuildPartialVR-Tree($n_i$);
           if $gn_i$ in not $null$
             then $g_n.childrenList.add(g_{n_i})$;
 else if node instanceof RTable
 then {$g_n \leftarrow$ AssignLabelTable($node.childrenList$)};
return ($gn$)
```

**Figure 1. BuildPartialVR-Tree algorithm for WGIs**

```
Algorithm 2 BuildFinalVR-Tree(GroupNode node)
L $\leftarrow$ node.childrenList;
if node was derived from RBlock or RTableCell
  then
    for each $n_i \in L$
      do {BuildFinalVR-Tree($n_i$);
    FindGroupsInBlock($L$);
 if node was derived from RTable
 then FindGroupsInTable($L$);
FindSuperGroups($L$);
```

**Figure 2. BuildFinalVR-Tree algorithm for WGIs**

groups/supergroups. Both are recursive. These algorithms are shown in full in figures 1 and 2.

These algorithms are evaluated using 20 WQIs from five domains: airfare, automobile, book, job, and real estate, selected at random from two datasets (ICQ and Tel-8). To evaluate, they use three metrics, including precision (correctly labeled fields vs labeled fields), recall (correctly labeled fields vs all fields), and the f-measure (harmonic mean of the previous factors). In doing so, they demonstrate these algorithms have an average f-measure of approximately 95%. This sometimes outperforms several competing algorithms while requiring less domain-specific knowledge.

### 1.2 Supergraph Search in Graph Databases via Hierarchical Feature-Tree [5]

Hierarchical feature-trees can be used to optimize supergraph search on graph databases. Supergraph search on Q is an algorithm for retrieving all data graphs contained in Q within the database. This technique is useful for many fields, including chemistry, biology, and computer vision. Typically, it involves subgraph isomorphism testing, an NP-complete problem; inclusion and exclusion logic can reduce complexity but not remove the NP-completeness, limiting scalability. To improve upon this, the graph database is typically pruned using features. Some features are "frequent features"; many of these are present in Q and thus cannot be pruned, leading to computation time wasted on these features.

Thus, a new method is proposed using hierarchical feature-trees called "DGTrees." The tree is organized such that each feature is a subgraph of its descendants (increasing complexity lower in the tree). When the upper levels fail to prune, the descendants are used. DGTrees consist of tree-nodes. These tree-nodes are a data structure containing the eight following elements

```
Algorithm 1. DGTreeConstruct(database $\mathcal{D} = \{G_1, \ldots, G_n\}$)
$g_r \leftarrow$ a new tree-node;
$g_r$.graph $\leftarrow$ a single-edge graph;
$g_r.\mathcal{S} \leftarrow \mathcal{D}$; $g_r.\mathcal{S}^* \leftarrow \mathcal{D}$; $g_r$.grow-edge $= \emptyset$;
for $G_i \in \mathcal{D}$ and $(v, v') \in E(G_i)$ do
  $g_r.\mathcal{M}(G_i) \leftarrow g_r.\mathcal{M}(G_i) \cup \{[v, v'], [v', v]\}$;
TreeGrow($g_r$);
return $g_r$;
Procedure TreeGrow(tree-node $g$)
$\mathcal{H} \leftarrow$ CandidateFeature($g$);
$\mathcal{C} \leftarrow g.\mathcal{S}^*$;
while $\mathcal{C} \neq \emptyset$ do
  $g^+ \leftarrow$ BestFeature($\mathcal{H}, \mathcal{C}$);
  if $|g^+.\mathcal{S}^*| > 1$ then
    $g^+$.graph $\leftarrow$ a graph by adding $g$.grow-edge in $g$.graph;
    TreeGrow($g^+$);
  else $g^+$.graph $\leftarrow$ the graph in $g.\mathcal{S}^*$; $g^+.\mathcal{S} \leftarrow g^+.\mathcal{S}^*$;
  $g$.children $\leftarrow g$.children $\cup \{g^+\}$;
  $\mathcal{C} \leftarrow \mathcal{C} \setminus g^+.\mathcal{S}^*$;
```

**Figure 3. DGTreeConstruct algorithm for supergraph search**

1. g.children: a list of child nodes.
2. g.graph: the graph (nodes and vertices) that it represents.
3. grow-edge: an edge not present in its parent node.
4. edge-type: OPEN if a new node is added not in the parent; CLOSE otherwise.
5. g.S: a set of graph-names containing its graph as a subgraph.
6. g.M(G_i): a set of matches of the graph.
7. g.S*: a set of nodes such that all child nodes' S* form a disjoint cover of g.S*.
8. g.score: a score used to select the best edge to grow.

Using these elements, nine algorithms are used to create, manipulate, and optimize a DGTree. For our purposes, we shall focus mostly on the first three algorithms, used for indexing. These 3 are shown in figures 3, 4, and 5.

The (1) DGTreeConstruct algorithm references (2) CandidateFeature, which generates candidate features. From these features, (3) BestFeature selects the one with the highest score to use for a new node. There are multiple score functions that can be used for selecting the node, with the simplest of these being the magnitude of g.S*; this maximizes for graph coverage by the tree but without taking into account pruning power. The second (more complicated) score function corrects this by taking the number of matches (g.M) into account. These algorithms combined form a heuristic approach to indexing with a non-exponential growth, which does not require isomorphism checking, and which allows infrequent features (often with high pruning power).

Once the tree is constructed and used for indexing, algorithms 4 and 5 are used to complete a supergraph search. Algorithm 6 is used for optimization, while algorithms 7 and 8 allow for dynamic, online support, allowing nodes to be inserted or deleted from an existing database without disrupting the

**Algorithm 2.** CandidateFeature(tree-node $g$)

```
$\mathcal{H} \leftarrow \emptyset$;
  **for** data-graph $G \in g.\mathcal{S}^*$ **and** match $f \in g.\mathcal{M}(G)$ **do**
    **for** $u_i \leftarrow 1$ **to** $|f|$ **and** $v \in$ Nbr$(f(u_i), G)$ **do**
    **if** $v \in f$ **then**$u_j \leftarrow f^{-1}(v)$; $t \leftarrow$ CLOSE;
      **else** $u_j \leftarrow |f| + 1$; $t \leftarrow$ OPEN;
      **if** $u_j > u_i$ **and** $(u_i, u_j) \notin E(g)$ **then**
        $g^+ \leftarrow \mathcal{H}$.Find$((u_i, u_j))$;
        **if** $g^+ = \emptyset$ **then**
          $g^+ \leftarrow$ a new tree-node;
          $g^+$.grow-edge $\leftarrow (u_i, u_j)$;
          $g^+.\mathcal{S}^* \leftarrow \{G\}$; $g^+$.score $\leftarrow 0$;
          $g^+$.edge-type $\leftarrow$ t;
          $\mathcal{H}$.Push$(g^+)$;
      **else** $g^+.\mathcal{S}^* \leftarrow g^+.\mathcal{S}^* \cup \{G\}$;
**for** data-graph $G \in g.\mathcal{S}$ **and** match $f \in g.\mathcal{M}(G)$ **do**
  **for** $u_i \leftarrow 1$ **to** $|f|$ **and** $v \in$ Nbr$(f(u_i), G)$ **do**
    **if** $v \in f$ **then**$u_j \leftarrow f^{-1}(v)$;
    **else** $u_j \leftarrow |f| + 1$;
    **if** $u_j > u_i$ **and** $(u_i, u_j) \notin E(g)$ **then**
      $g^+ \leftarrow \mathcal{H}$.Find$((u_i, u_j))$;
      **if** $g^+ \neq \emptyset$ **then**
        $g^+.\mathcal{S} \leftarrow g^+.\mathcal{S} \cup \{G\}$;
        **if** $g^+$.edge-type $=$ OPEN **then**
          $g^+.\mathcal{M}\,(G) \leftarrow g^+.\mathcal{M}\,(G) \cup \{[f, v]\}$;
        **else**$g^+.\mathcal{M}\,(G) \leftarrow g^+.\mathcal{M}\,(G) \cup \{f\}$;
**for** $g^+ \in \mathcal{H}$ **do**
  compute $g^+$.score; $\mathcal{H}$.Update$(g^+)$;
**return** $\mathcal{H}$;
```

**Figure 4. CandidateFeature algorithm for supergraph search**

**Algorithm 3.** BestFeature(heap $\mathcal{H}$, uncovered graphs $\mathcal{C}$)

```
$g^+ \leftarrow \mathcal{H}$.Pop();
**while** $g^+.\mathcal{S}^* \not\subseteq \mathcal{C}$ **do**
  $g^+.\mathcal{S}^* \leftarrow g^+.\mathcal{S}^* \cap \mathcal{C}$;
  **if** $g^+.\mathcal{S}^* \neq \emptyset$ **then**{compute $g^+$.score; $\mathcal{H}$.Push$(g^+)$};
  $g^+ \leftarrow \mathcal{H}$.Pop();
**return**$g^+$;
```

**Figure 5. BestFeature**

search indexing. Lastly, algorithm 9 is used to reduce false negatives, thereby increasing efficiency.

To evaluate the efficiency of this method, it was compared against the IGquery and PrefIndex algorithms (two preexisting competing algorithms) on the CCD and NCI databases. The results showed that for larger databases and denser graphs, DGTree outperforms the competitors. It remains competitve in simpler cases (but sometimes performs worse). This demonstrates the scalability brought on by the approximations that allow for non-exponential calculation.

### 1.3 A Survey of B-Tree Logging and Recovery Techniques [3]

First, a distinction is noted between logical and physical contents of trees; two trees representing equivalent data can differ if the orders of insertion differ. Logical data is protected by locks, while semaphores protect critical areas of physical memory. This leads to a difference in user and system transactions and in logical, physical, and physiological logging. Transactions are independent, atomic, consistent, and durable actions affecting a database structure and its contents. They are typically handled by a thread pool with threads that compete for locks.

The basis of logging is "write-ahead logging." Each change has a "do" method for starting a logged change and "undo" and "redo" methods for reverting or repeating a change in the event of a failure. Transaction failures involve an application cancelling a transaction (perhaps automatically due to deadlock or lack of space); these must revert a single change. A media failure is due to communication failure with a hard disk (perhaps removed from the system), while a system failure results from a crash or outage of the operating system or hardware. These failures requires repeating a sequence of logged changes. Frequently, a checkpoint should be added to the log consisting of changes within an interval; this shortens recovery time for media and system failures.

Logging should be stored on a stable but slower storage, while database transactions should be optimized for speed even with greater risk of failure. Failure can corrupt even completed transactions if they were written to cached data that hasn't been written back. A transaction should not be logged as completed until written back. On shutdown, the last log entry should be a checkpoint noting no active transactions.

Devices with limited memory can employ group commit: delaying log entries until a memory page is full. Devices can also create checkpoints of unfinished transactions, dirty cache pages, etc. to speed up recovery; these should point to the last completed log record and the last log record to undo. Each log entry should have a Log Sequence Number (LSN) for unique identification and for quick access via hashing.

User transactions are requested through an application. Reversal (undo) must be possible until the application commits the changes. Once committed, the changes must persist through failure (via redo). Multiple forms of undo exist including erasing the record, marking it invalid, or performing the opposite action (updating back). Undo is logical and may not reverse all physical changes.

User transactions can utilize lower-level (potentially nested) system transactions. User transactions modify the logical structure (inserts, deletes, etc.) while system transactions modify the physical structure (splitting/merging nodes, etc.). System transactions inherit locks and other context from a user transaction that calls them. System transactions are not logged except in the context of user transactions that require their completion.

Physical logging logs the change in memory, often as a before-state and list of modified bytes, perhaps compressed with a run-length encoding. Logical logging records operations taken but not their precise memory locations. Logical logging requires less space but is more complicated to restore. Physiological logging is a compromise, where log entries refer to a page number (physical) and a record (logical) within that page.

Deleted records can be marked as invalid "ghost records." These reduce deletion time by allowing the system to overwrite memory later. Recovery is likewise easy (just reset the invalid bit) and less is logged (one bit vs. a whole record). Logical insertions of identical keys should reuse invalid keys rather than create a duplicate. Page compaction removes ghost records for space compression.

Several additional optimizations exist to reduce log complexity of undo and redo operations. Ghost record creation and replacement can be fused into a single record if these occur closely in time. Careful write ordering involves logging operations and relevant parameters rather than entire database records. The source of a copy operation is preserved until copied to the destination; it acts as the destination's backup and as such prevents a need for a logged backup.

## 2 QUERYING

Being able to parse through large data is essential to understanding the knowledge at hand. However, big data sets make it difficult to sift through and sort timely and accurately. Using algorithms is then necessary to not only query the information, but to also make it a much more quick and reliable search.

### 2.1 Fast Algorithms for Mining Association Rules [1]

Bar-code technology has made it easier for companies to store large amounts of data. The data can consist of the name of the item, date purchased, purchased price, etc.. This is important to keep track of what is coming in and out, as well as knowing what sells. This is important for marketing teams to create strategies and plans such as catalog design, add-on sales, and store layout based off customer buying patterns. Fast algorithms are needed to carry out these databases that contain a large amount of information.

The following is a formal statement of the problem: Let $I = i_1, i_2, ...i_m$ be a set of literals, called items. Let $D$ be a set of transactions, where each transaction $T$ is a set of items such that $T \subseteq I$. Associated with each transaction is a unique identifier, called its $TID$. We say that a transaction $T$ contains $X$, a set of some items in $I$, if $X \subseteq T$. An association rule is an implication of the form $X \rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = 0$. The rule $X \rightarrow Y$ holds the transaction set $D$ with confidence c if c% of transactions in $D$ that contain $X$ also contain $Y$. The rule $X \rightarrow Y$ has support $s$ in the transaction set of $D$ if s% of transactions in $D$ contain $X \cup Y$.

Once given the set of transaction, we can generate the rules that have support and confidence greater than the user-specified thresholds, called "minsup" and "minconf" respectively. There had previously been two algorithms to find all association rules called AIS algorithm and SETM algorithm. This paper presents a new algorithm called Apriori that differ fundamentally from the first two mentioned. Not only does new algorithms outperform the original two, but the performance gap increases with problem size.

Discovering all association rules can be difficult and the matter can be separated into two subproblems. The first is finding all itemsets that have transaction support above minimum support. The support of an itemset is the number of transactions that contain the itemset. The second is using the itemsets to create rules. This paper however only addresses in solving the first problem.

In order to find large itemsets, the algorithm will make multiple passes over the data. The first pass will consist of counting the support of individual items to determine which of them have the minimum support. In each additional pass, a seed set of large itemsets in the past round are used to find potential new large itemsets, called candidate itemsets. This continues until no new large itemsets are found.

In the original algorithms, the candidate itemsets are generated impulsively during the pass as data is being read. The Apriori algorithm instead generate the candidate itemsets to be counted in a pass by using only the itemsets found large in the previous pass. This results in a much smaller passed set.

The Apriori algorithm's first pass counts item occurances to determine large 1-itemsets. The next pass will consist of two parts. The first part will use the large itemset found in the previous round to create candidate itemsets using the following algorithm. The second part will consist of scanning the database and counting the support of the candidate items set found in part one of this pass.

Candidate itemsets are stored in a hash-tree. A node of the hash-tree either contains a leaf node of a list of itemsets or an interior node of a hash table. In the hash table, each bucket of the hash table points to another. However, all nodes are initially created as leaf nodes. It is only when then number of itemsets in a leaf node exceeds a specified threshold that the leaf node is converted to an interior node.

When starting at the root node, the subset function finds all the candidates contained in a transaction, t. This is because when we are at a leaf node, we find which of the itemsets in the leaf are contained in t and add references to them to the answer set.

When it came to performance, this new method outperformed the previous methods every time. The performance gap even increased with the problem size, showing that this can be easily scaled.

### 2.2 Exploiting the Potential of Large Databases of Electronic Health Records for Research using Rapid Search Algorithms and an Intuitive Query Interface [9]

The growth of technology has led to electronically kept health records. This information is of great resource for those in the health services and clinical research industry, as these records can help to identify patients with a specific disease or condition and investigate patterns of diagnosis and symptoms.

The UK's primary care database system contains diagnostic, demographic, and prescribing information for millions of patients. However, extracting relevant information can be difficult and time-consuming, since this database contains information about millions of patients and each patient can have multiple records.

This new algorithm can be used to identify patients that are fit into the specification for further screening for recruitment into randomized controlled trials within general practices. TrialViz would parse the data that is held within The Clininal Practice Research Datalink (CPRD), which represents the largest collection of anonymized primary care patient records in the world, to enable users to select General Practices (GP) based on suitability of the patient base for the intended study and practice-based measures of the quality of data recording. Demographic and clinical parameters will further help parse the data.

Based off feedback from a team of data analysts, epidemiologists, statisticians, graphical designers, software engineers, and computer scientists, the following were chosen as particularly important for this new engine.

1. Data abstraction tools – users can upload or select codes to create rules or queries for selecting the patients of interest
2. Data extraction tools for running queries using the data abstraction riles in close to real time
3. A protocol for measuring data quality for each practice and how fit it is for a particular study
4. Visualization tools to investigate the results

A challenge that is presented when parsing through the data is the amount of data present. As mentioned earlier, each patient may have multiple records, and there may also be multiple codes for a disease or symptom. For example, there are over 200 codes for diabetes and 40 codes that align with abdominal pain. After running these queries, another challenge that is presented is the process time involved.

SQL is known to not handle large datasets well. So instead, an interface based on 'stacks and cards' was developed for the web portal to run complex queries interactively and to visualize the results. Each card represents the results of a single query and the stacks represent a container of cards in which users can place multiple cards. Furthermore, the stack serves two purposes. The first purpose is to have a visual presentation for the set theory rules. The second purpose is to select the date ranges. A search is built using these cards and cards are essentially patient lists. Cards are logically grouped on stacks and stack represents a union of the card it contains while the search represents the intersect of the participating stacks.

$$f(K) = \begin{cases} 1 & \text{if } K = 1 \\ \dfrac{S_K}{\alpha_K S_{K-1}} & \text{if } S_{K-1} \neq 0, \forall K > 1 \\ 1 & \text{if } S_{K-1} = 0, \forall K > 1 \end{cases} \quad (2)$$

$$\alpha_K = \begin{cases} 1 - \dfrac{3}{4N_d} & \text{if } K = 2 \text{ and } N_d > 1 \quad (3a) \\ \alpha_{K-1} + \dfrac{1 - \alpha_{K-1}}{6} & \text{if } K > 2 \text{ and } N_d > 1 \quad (3b) \end{cases}$$

**Figure 6. K-Cluster Evaluation Function**

### 2.3 Selection of K in K-means clustering [7]

Clustering is an important part of different applications such as data mining and knowledge discovery, data compression and vector quantization, and pattern recognition pattern classification. It allows objects with similar characteristics to be grouped together to make processing the data easier. The K-means algorithm is a well-known data clustering algorithm, but to use it requires a pre-known and user chosen number of clusters. However, the challenges that arise with this is that it can be subjective in nature of deciding what that number should be.

When determining K, the number must be reasonably large but significantly smaller than the number of objects in the data set. This will create clusters of substantial sizes. Previous studies of K-means clustering did not specify any logic for choosing a value for K. Previous mistakes that had been made are a) using one or two values for K and b) using a relatively large K in comparison to the number of objects. Often times, this number is found by trial-and-error.

Other than there being no set method in choosing K, there are also other outside factors. For example, the level of detail required. A data set with n objects grouped into clusters between 1 and n would respectively correspond to lowest and highest level of details. This is because in order to further group the data, one would need more information about the observed objects.

To evaluate the cluster result, a function $f(K)$ is used. The evaluation function is defined in Figure 6 where $S_k$ is the sum of the cluster disortions when the number of clusters is $K$, $N_d$ is the number of data set attributes (such as the number of dimensions) and $\alpha_k$ is a weight factor.

This new function of $f(K)$ can allow for multiple suggestions of K for various levels of details. It also takes into account information reflecting the performance of the algorithm.

## 3 BIG DATA

Big data queries are a particular difficulty for database design in traditional implementations. The serial access as well as the large size of the types of queries executed, sometimes larger than the available memory on the machine, makes access time, reliability, durability, and the time frame for gathering and interpreting useful information unreasonable. Augmenting this process with distributed and replicated database systems improves upon these issues but has the added complexity of maintaining proper access patterns and predictability when using the system. Algorithms must be specifically designed to either address these issues or augment previously used techniques in order to work with such a setup. In some cases however, parallelization may make some traditional algorithms more feasible for use on distributed systems, specifically in the context of overhead.

### 3.1 An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases [2]

Replicated/distributed DBs add additional complexity in terms of security and consistency of operations in comparison to traditional DBs. The complexity is worth it for the benefits but this is only if the correctness of operations is maintained. Replication and concurrency control must be guaranteed in order for correctness to be.

Replication control means that a set of replicas behaves the same as a single unit would. An operation done on any set of copies must affect the entire set of replicas as if they were one. Concurrency control means the effect of a series of executions on a traditional, single database system must be the same for distributed systems, where they may be running in parallel and thus, not necessarily in order.

To handle this, the typical method would be two-phase locking (2PL). This means that a read is performed on any copy of the data, writes update all copies of the data they are changing, and concurrency control is handled with locks on particular copies during and depending on access Reads acquire read-locks and writes acquire write-locks.

This addresses the problem of concurrency control but not of replication control which the proposed algorithm attempts to solve. For "an environment where sites fail and recover," a method of being able to dynamically react to replica sites being up or down is required. This ensures that the physical execution of transactions on the multiple replicas matches the logical order/execution and expected effect of the submitted, declarative-style transaction.

This is accomplished mainly by implementing data directories. These are created for each item in the database and store (1) a list of all the available copies of the data item and (2) a list of all the available copies of the directory for that item. This means that the directories themselves must also be replicated.

These directories are used for status transactions, which are used to update availability and access meta-data for each data item. Include() and Exclude() are used to make a copy available/unavailable for user transactions, based on the availability of the site storing that particular copy. There is also a DirectoryInclude() status transaction which performs the same function but for a particular copy of an item's directory. It is described that a DirectoryExclude() is not necessary since a directory copy immediately becomes unavailable when its site fails. Additional locks are used to change values in the directories, consistent with 2PL.

With the status transactions available, read() and write() calls can be changed to fit with the new system goals. Each user transaction has a transaction manager, which handles its meta-data related status transactions and acquires an available directory copy for that data item. An available directory copy is now required for any transaction to proceed. Having no available directory copies implies that every site for that particular data item is down and that the user transaction cannot occur.

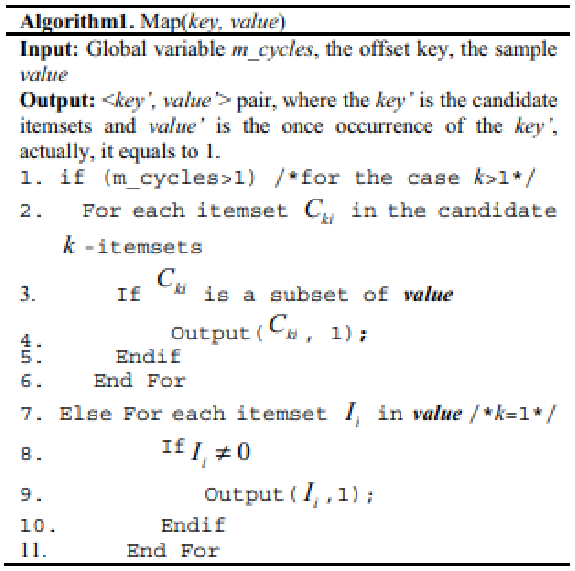
With a directory copy available, reads and writes can occur on data items. Reads are expanded to set a read-lock on some available data copy from the directory copy's list of available data items, and read the data from that copy. This is in contrast to simply setting a read-lock on any copy and reading from it. Writes are a little bit more complicated. Before setting the write-lock on each of the available data copies, a lock must be set on the directory copy being used. Once both of these are complete, the write-locked copies are checked once again for availability. The data copy is written to if it is still available and ignored otherwise.

Between the first and second check for availability in the write procedure, the transaction reaches its locked point, which is when all of the data copies in the directory for that item are either write-locked or unavailable. This is used for cleanup of the transaction. The transaction is aborted, since it cannot ensure correctness, if there is any mismatch between the data copies that were read and the available data copies still in the directory copy's list or if the particular data item is currently being excluded. The process for this is also parallelizable, making it reasonably fast.

In order for these implementations to be possible, the algorithm also makes some assumptions about the predictability and type of failures. The failures must be clean, meaning that when a site goes down it does so completely, not in a way in which it still seems to respond but performs transactions incorrectly. Network failures are assumed to be the same, where the network is either completely up or down. Additionally, it is assumed that the network and the sites themselves handle recovery and failure reporting, with the algorithm implementation only needing to check up/down status. Additionally, the algorithm performs best when site failures are uncommon and "data access patterns [are] predictable." When all assumptions are met, the algorithm is believed to be a preferable alternative to 2PL, although a correctness proof is only sketched out and not formally performed.

### 3.2 Parallel Implementation of Apriori Algorithm Based on MapReduce [4]

Algorithms that perform transactions on databases are not the only ones that require adjustment when considering distributed databases. Access patterns, such as frequent access, and the algorithms behind them also need to adjust depending on the scale of the database, especially in cases where dataset size is larger than memory. The typical way to do this is to use the Apriori algorithm, described in section 2.1. However, since the calculation overhead of apriori scales with database size and makes it decreasingly effective as database size increases. The



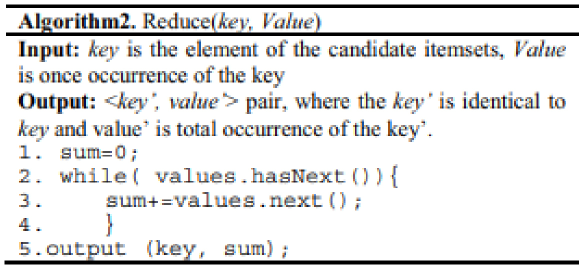
**Algorithm1.** Map(*key, value*)

**Input:** Global variable *m_cycles*, the offset key, the sample *value*

**Output:** <*key', value'*> pair, where the *key'* is the candidate itemsets and *value'* is the once occurrence of the *key'*, actually, it equals to 1.

```
if (m_cycles>1) /*for the case $k$>1*/
    For each itemset $C_{ki}$ in the candidate $k$-itemsets
        If $C_{ki}$ is a subset of value
            Output($C_{ki}$, 1);
        Endif
    End For
Else For each itemset $I_i$ in value /*$k$=1*/
        If $I_i \neq 0$
            Output($I_i$,1);
        Endif
    End For
```

Figure 7. PApriori Map Step



**Algorithm2.** Reduce(*key, Value*)

**Input:** *key* is the element of the candidate itemsets, *Value* is once occurrence of the key

**Output:** <*key', value'*> pair, where the *key'* is identical to *key* and value' is total occurrence of the key'.

```
sum=0;
while( values.hasNext()){
    sum+=values.next();
    }
output (key, sum);
```

Figure 8. PApriori Reduce Step

proposed algorithm by Li et. al. recognizes the effectiveness of the frequent-access pattern that Apriori provides and combines this algorithm with MapReduce for calculation steps in order to reduce the overhead of the calculation required to maintain the access pattern. It is easy to see how an algorithm like Apriori, which starts with a large dataset of small items and combines them into a small result set, can be augmented with MapReduce, which is used to parallelize such computations.

The actual proposed PApriori (for parallel Apriori) algorithm considers the use of MapReduce at the combination step for Apriori computations. Starting with the candidate itemsets of size 1, MapReduce is run iteratively for the combination of associations. This is possible because the individual occurrence counting of itemsets in transactions has no overlap, causing no concurrency issues, which would add additional overhead. The map step (figure 7) takes the candidate itemsets and outputs them, mapped by transaction to their respective reduce nodes, based on their membership of said transactions. The reduce nodes (figure 8) then combine these results, summing the values for the current transaction, and output the results to the next iteration of the PApriori process, so that the strong rules for the k-itemsets can be generated. This is done on every iteration until the specified, maximum number of iterations has been reached (K < maxIterations). This means that the most intensive portion of the Apriori calculation receives a boost in performance due to the parallelization offered by MapReduce.

This proposed performance increase is consistent with the results prescribed in the paper. Evaluation is performed using scaleup: "the ability of an m-times larger system to perform an m-times larger job in the same time as an original system," sizeup: "how much longer it takes on a given system when the dataset size is m-times larger than the original dataset," and speedup: "how much [the] parallel algorithm is faster than" its corresponding sequential counterpart. For small datasets, as expected, the performance does not scale well, especially with higher CPU-core counts. For larger datasets, however, performance is increased particularly well, with the relationship of core count to speedup becoming more and more linear with increasing slope for the largest dataset sizes tested. When datasets have " either many short transactions with few frequent itemsets, or fewer larger transactions with many frequent itemsets, PApriori algorithm [shows] good performance." This result is also particularly important for the same reasons that MapReduce is, in that this speedup is achieved using "commodity hardware" rather than highly complex, powerful, and expensive single computing systems.

## 3.3 SQL: From Traditional Databases to Big Data [8]

SQL is popular because it is declarative and simple to learn. The simplicity is an important aspect that makes SQL accessible to a vast number of people but the declarative nature shines since it is able to provide opportunities for extraction and optimization. This makes it useful for newer types of databases which use new technologies to speed up queries, access, and implementation. The ways in which this can be done are organized into three technologies: MapReduce, NoSQL, and NewSQL.

MapReduce works by splitting workload into chunks, mapped to different reducer nodes, reduced to final answers, and combined to get the final answer(s). Instead of requiring a single machine to perform the entire operation on a certain set, the workload is distributed among multiple machines. This not only makes it easier to achieve desired runtimes, but also more accessible, since each machine (worker node) only needs basic hardware to perform its portion of the workload.

In a Mapreduce setup the dataset for the operation is divided into chunks which are portioned out (mapped) to specific reducer nodes. These reducer nodes perform their operation on (reduce) their subset of the data and return an answer to one or more combine nodes which assemble (combine) the individual results from the reduce step to produce the final result(s) of the operation. This can be used for database queries as the control flow of the actual physical operations on the data, with SQL as the logical operation "front-end" for the system.

NoSQL is a non-relational database system, intentionally built to be scalable and distributable. It uses key-value pairs and documents to store data in such a way that organization is achieved through nesting of these pairs or documents. Because of this, the system is schema-free and has little need for join operations, since data can be reached by simply searching the documents for the data needed.

It is also very scalable and highly available. The insertion of documents based on context means that it does not have to be planned ahead of time, and expands as needed. The system does not have to be redesigned when the data changes since no schema needs to be changed. SQL is also easy to implement for it, despite the name, because of SQL's declarative property. SQL operation control flow can be specifically implemented and adapted to each specific use case in order to allow the logical operations typical of SQL to perform as expected. New SQL is described to take this last principle and recreate it as a standalone system, designed to be a modern, NoSQL-backed implementation of traditional, SQL-style relational databases. This makes it much faster and retains the modern accessibility and scalability attributes.

## CONCLUSION

Database algorithms that work on top of the database system are generally used for improving performance and decreasing access time. Indexing as a category acts as a general speed increase as well as allowing concurrent access at times when different parts of data are accessed. Querying specialization algorithms are used to shape the desired data in such a way that performance is optimized for specific use cases, whether catering for frequent access, grouped information, or large queries. Further enhancing specifically those queries that are very large, distributed systems allow for parallelized access of desired data which improves performance but adds additional complexity. Algorithms used for indexing and specific use case queries have to be adjusted to work with distributed systems but can also be much faster due to the parallel access reducing the overhead involved with these algorithms. However, additional techniques also have to be created in order to help ensure what is normally ACID compliance, but applied to distributed systems in the form of BASE compliance. These algorithms further rely on the principles of indexing and use case querying optimization to ensure a reasonably small overhead for these additional operations.

## REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 487–499.

[2] Philip A. Bernstein and Nathan Goodman. 1984. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984), 596–615. DOI: http://dx.doi.org/10.1145/1994.2207

[3] Goetz Graefe. 2012. A Survey of B-Tree Logging and Recovery Techniques. *ACM Transactions on Database Systems* 37, 1 (2012), 1 – 1:35. http://proxy01.its.virginia.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=iih&AN=72956519&site=ehost-live&scope=site

[4] Ning Li, Li Zeng, Qing He, and Zhongzhi Shi. 2012. Parallel Implementation of Apriori Algorithm Based on MapReduce. In *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. 236–241. DOI: http://dx.doi.org/10.1109/SNPD.2012.31

[5] Bingqing Lyu, Lu Qin, Xuemin Lin, Lijun Chang, and Jeffrey Xu Yu. 2019. Supergraph Search in Graph Databases via Hierarchical Feature-Tree. *IEEE Transactions on Knowledge Data Engineering* 31, 2 (2019), 385 – 400. http://proxy01.its.virginia.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=iih&AN=134073130&site=ehost-live&scope=site

[6] Heidy Marin-Castro and Victor Sosa Sosa. 2017. VR-Tree: A novel tree-based approach for modeling Web Query Interfaces. *Journal of Intelligent Information Systems* 49, 3 (2017), 367 – 390. http://proxy01.its.virginia.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=iih&AN=126055153&site=ehost-live&scope=site

[7] D T Pham, S S Dimov, and C D Nguyen. 2005. Selection of K in K-means clustering. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 219, 1 (2005), 103–119. DOI: http://dx.doi.org/10.1243/095440605X8298

[8] Yasin N. Silva, Isadora Almeida, and Michell Queiroz. 2016. SQL: From Traditional Databases to Big Data. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 413–418. DOI: http://dx.doi.org/10.1145/2839509.2844560

[9] A. Rosemary Tate, Natalia Beloff, Balques Al-Radwan, Joss Wickson, Shivani Puri, Timothy Williams, Tjeerd Van Staa, and Adrian Bleach. 2014. Exploiting the potential of large databases of electronic health records for research using rapid search algorithms and an intuitive query interface. *Journal of the American Medical Informatics Association* 21, 2 (2014), 292 – 298. http://proxy01.its.virginia.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=iih&AN=94425993&site=ehost-live&scope=site