Berkeley Databases Prelim Notes

Shreya Shankar

Summer 2022

This document served as my study guide for the Berkeley DB prelim (syllabus here). Much of the text is copy-pasted from the original papers. Some of the text is copy-pasted from Michael Whittaker's notes and self-contained paper summaries, both of which I am very grateful for. A miniscule fraction of the text is copy-pasted from old CS262A and CS286 notes, scattered across the internet.

My contribution is mainly the comprehension questions, which are spread throughout the document. Although I spent several months[1] studying, I did not get to every subsection in each paper.

[1] I would not have been able to do this without the support of my partner, who made me chai or coffee almost every day.

Table of Contents

1 Basics and Foundational Systems **14**

1.1 Architecture of a Database System [33] 14

1.1.1 Process Models 16

1.1.2 Parallel Architecture 18

1.1.3 Relational Query Processor 20

1.1.4 Storage Management 23

1.1.5 Transactions: Concurrency Control and Recovery 25

1.1.6 Shared Components 27

1.2 The Five Minute Rule Twenty Years Later [25] 28

1.3 A History and Evaluation of System R [13] 29

1.3.1 Phase Zero 30

1.3.2 Phase One 30

1.3.3 Phase Two 31

1.4 The POSTGRES Next-Generation Database System [49] 33

1.4.1 The POSTGRES Data Model and Query Language 34

1.4.2 The Rules System 34

1.4.3 The Storage System 35

1.5 The Gamma Database Machine Project [19] 36

1.5.1 Software Architecture of Gamma 39

1.5.2 Query Processing 40

1.5.3 Transaction and Failure Management 41

1.5.4 Lessons Learned 42

2 Query Processing **44**

2.1 Access Path Selection in a Relational Database Management System [47] . . 44
2.1.1 Processing of a SQL Statement . . . 45
2.1.2 Research Storage System . . . 45
2.1.3 Costs for Single Relation Access Paths . . . 46
2.1.4 Access Path Selection for Joins . . . 48
2.1.5 Nested Queries . . . 49
2.2 Query Evaluation Techniques for Large Databases [24] . . . 49
2.2.1 Architecture of Query Execution Engines . . . 50
2.2.2 Sorting . . . 51
2.2.3 Hashing . . . 52
2.2.4 Disk Access . . . 53
2.2.5 Aggregation and Duplicate Removal . . . 54
2.2.6 Binary Matching Operators . . . 55
2.2.7 Universal Quantification . . . 56
2.2.8 Duality of Sort and Hash . . . 57
2.2.9 Execution of Complex Query Plans . . . 58
2.2.10 Mechanisms for Parallel Query Execution . . . 58
2.2.11 Parallel Algorithms . . . 60
2.2.12 Nonstandard Query Processing Algorithms . . . 61
2.3 The Volcano Optimizer Generator: Extensibility and Efficient Search [26] . . 61
2.3.1 Optimizer Generator Input and Optimizer Operation . . . 62
2.3.2 The Search Engine . . . 63
2.3.3 Comparison with the EXODUS Optimizer Generator . . . 64
2.4 Eddies: Continuously adaptive query processing [6] . . . 66
2.4.1 Ripple Joins . . . 67

2.4.2 Rivers and Eddies . . . 68
2.4.3 Routing Tuples in Eddies . . . 68
2.4.4 Responding to Dynamic Fluctuations . . . 69
2.4.5 Advantages and Disadvantages of Eddies . . . 69
2.5 Worst-Case Optimal Join Algorithms [42] . . . 70
2.6 Datalog and Recursive Query Processing—Sections 1-3 and 6 [28] . . . 71
2.6.1 Baby's First Example . . . 72
2.6.2 Language and Semantics . . . 72

3 Transactions **74**

3.1 Transaction Basics [45] . . . 74
3.1.1 Serializability . . . 75
3.1.2 Lock-based Concurrency Control . . . 76
3.1.3 Isolation Levels . . . 77
3.1.4 2PL, Serializability, and Recoverability . . . 77
3.1.5 Lock Conversions and Deadlocks . . . 79
3.1.6 Concurrency Control in B+ Trees . . . 80
3.1.7 Multiple-Granularity Locking . . . 80
3.1.8 Optimistic Concurrency Control . . . 81
3.1.9 Timestamp-Based Concurrency Control . . . 83
3.1.10 Multiversion Concurrency Control . . . 84
3.2 ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging [41] . . . 85
3.2.1 The Log . . . 86
3.2.2 Other Recovery-Related Structures . . . 87
3.2.3 Checkpointing . . . 88

3.2.4 Executing the Algorithm . . . 88
3.2.5 Interaction with Concurrency Control . . . 89
3.2.6 System R Recovery Algorithm . . . 89
3.2.7 Parallelism . . . 90
3.3 Granularity of locks and degrees of consistency in a shared data base [27] . . . 91
3.3.1 Granularity of Locks . . . 92
3.3.2 Degrees of Consistency . . . 93
3.3.3 Dependencies Among Transactions . . . 94
3.3.4 Backups and Recovery . . . 94
3.4 Concurrency Control in Distributed Database Systems [8] . . . 94
3.4.1 Decomposition of the Concurrency Control Problem . . . 96
3.4.2 Synchronization techniques based on 2PL . . . 96
3.4.3 Synchronization techniques based on T/O . . . 98
3.4.4 Integrated Concurrency Control Methods . . . 100
3.5 Concurrency Control Performance Modeling: Alternatives and Implications [3] . . . 100
3.5.1 Performance Model . . . 101
3.5.2 Performance Metrics . . . 101
3.5.3 Experiments . . . 102

4 Indexing **104**

4.1 Efficient Locking for Concurrent Operations on B-Trees [37] . . . 104
4.1.1 Storage Model . . . 104
4.1.2 Problem with Concurrent B+ trees . . . 105
4.1.3 B-link Trees . . . 106
4.1.4 Search . . . 106

4.1.5 Insertion . . . 106
4.1.6 Deletion . . . 107
4.2 Improved Query Performance with Variant Indexes [43] . . . 107
4.2.1 Indexing Definitions . . . 108
4.2.2 Comparing Index types for Aggregate Evaluation . . . 109
4.2.3 Evaluating OLAP-style Queries (join a fact table with multiple dim tables) . . . 109
4.3 The R*-tree: an Efficient and Robust Access Method for Points and Rectangles [7] . . . 110
4.3.1 R-tree Variants . . . 110
4.4 The Log-Structured Merge-Tree (LSM-Tree) [44] . . . 112
4.4.1 The two component LSM-tree algorithm . . . 112
4.4.2 The multi-component LSM-tree . . . 114
4.4.3 Concurrency Control in the LSM-tree . . . 114
4.4.4 Recovery in the LSM-tree . . . 114

5 DBMS Architectures Revisited **116**

5.1 C-Store: A Column-Oriented DBMS [50] . . . 116
5.1.1 Data Model . . . 117
5.1.2 RS . . . 117
5.1.3 WS . . . 118
5.1.4 Updates and Transactions . . . 118
5.1.5 Recovery . . . 119
5.1.6 Tuple Mover . . . 120
5.1.7 Query Optimizer . . . 120
5.2 Hekaton: SQL Server's Memory-Optimized OLTP Engine [20] . . . 121

5.2.1 Design Considerations . . . 121
5.2.2 Storage and Indexing . . . 122
5.2.3 Programmability and Query Processing . . . 122
5.2.4 Transaction Management . . . 122
5.2.5 Durability and Recovery . . . 124
5.2.6 Garbage Collection . . . 124
5.3 Calvin: Fast Distributed Transactions for Partitioned Database Systems [52] 125
5.3.1 Deterministic Database Systems . . . 126
5.3.2 System Architecture . . . 127
5.3.3 Calvin with disk-based storage and Checkpointing . . . 127
5.4 Spanner: Google's Globally-Distributed Database [15] . . . 128
5.4.1 Implementation . . . 128
5.4.2 Data Model . . . 129
5.4.3 Concurrency . . . 129
5.4.4 Differences between Calvin and Spanner . . . 129
5.5 Building Efficient Query Engines in a High-Level Language [36] . . . 130
5.5.1 Example . . . 131
5.5.2 General Execution Workflow . . . 132
5.5.3 Optimizations . . . 132

6 Distributed Data, Weak Isolation, Relaxed Consistency 134

6.1 Transaction Management in the R* Distributed Database Management System [40] . . . 134
6.1.1 Properties of an Ideal Atomic Commit Protocol . . . 135
6.1.2 The Two-Phase Commit Protocol . . . 135
6.1.3 Hierarchical 2PC . . . 136

6.1.4 The Presumed Commit Protocol . . . 137
6.1.5 Deadlock . . . 139
6.2 Generalized Isolation Level Definitions [1] . . . 139
6.2.1 Previous Work . . . 139
6.2.2 Restrictiveness of Preventative Approach . . . 140
6.2.3 Database Model and Transaction Histories . . . 141
6.2.4 Conflicts and Serialization Graphs . . . 142
6.2.5 New Generalized Isolation Specifications . . . 143
6.2.6 Mixing Isolation Levels . . . 144
6.3 Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System [51] . . . 144
6.3.1 System Model . . . 145
6.3.2 Conflict Detection and Resolution . . . 145
6.3.3 Replica Consistency . . . 145
6.3.4 Write Stability and Commitment . . . 146
6.3.5 Storage System Implementation Issues . . . 147
6.3.6 Access Control . . . 147
6.4 Dynamo: Amazon's Highly Available Key-value Store [17] . . . 147
6.4.1 System Interface . . . 148
6.4.2 Partitioning Algorithm . . . 148
6.4.3 Replication . . . 148
6.4.4 Data Versioning . . . 149
6.4.5 Execution of get () and put () operations . . . 149
6.4.6 Handling Failures . . . 149
6.4.7 Membership and Failure Detection . . . 150

6.4.8 Experiences and Lessons Learned . . . 150
6.5 CAP Twelve Years Later: How the "Rules" Have Changed [9] . . . 151
6.5.1 CAP-Latency Connection . . . 151
6.5.2 Managing Partitions . . . 152
6.5.3 Which operations should proceed? . . . 152
6.5.4 Partition Recovery . . . 152
6.6 Consistency Analysis in Bloom: a CALM and Collected Approach [4] . . . 152
6.6.1 BUD: Bloom Under Development . . . 153

7 Parallel Dataflow **155**

7.1 Parallel Database Systems: The Future of High Performance Database Processing [18] . . . 155
7.1.1 Trend to Shared Nothing DBs . . . 156
7.1.2 A Parallel Dataflow Approach to SQL Software . . . 156
7.1.3 Parallelism within Relational Operators . . . 157
7.1.4 State of the Art . . . 157
7.1.5 Future Directions . . . 157
7.2 Encapsulation of Parallelism in the Volcano Query Processing System [23] . 157
7.2.1 Bracket Model of Parallelism . . . 158
7.2.2 Volcano System Design . . . 158
7.2.3 Operator Model of Parallelization . . . 159
7.2.4 Horizontal Parallelism . . . 160
7.2.5 Variants . . . 160
7.3 MapReduce: simplified data processing on large clusters [16] . . . 161
7.3.1 Implementation . . . 161
7.3.2 Data Structures . . . 162

7.3.3 Refinements . . . 163
7.4 TAG: A tiny aggregation service for ad-hoc sensor networks [39] . . . 164
7.4.1 Motes and Ad-Hoc Networks . . . 164
7.4.2 Query Model and Environment . . . 165
7.4.3 In Network Aggregate . . . 166
7.4.4 Optimizations . . . 167
7.4.5 Loss . . . 168
7.5 Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing [53] . . . 168
7.5.1 Representing RDDs . . . 170
7.5.2 Memory Management . . . 170

8 The Web and Databases **172**

8.1 Combining Systems and Databases: A Search Engine Retrospective [10] . . . 172
8.1.1 SE Overview . . . 173
8.1.2 Logical Query Plan . . . 173
8.1.3 Query Implementation . . . 175
8.1.4 Implementation on a cluster . . . 176
8.1.5 Updates . . . 177
8.1.6 Fault Tolerance . . . 177
8.1.7 Other Topics . . . 177
8.2 The Anatomy of a Large-Scale Hypertextual Web Search Engine [11] . . . 178
8.2.1 System Features . . . 178
8.2.2 Anatomy . . . 179
8.2.3 Crawling . . . 180
8.2.4 Indexing . . . 180

8.2.5 Search . . . 180
8.3 WebTables: Exploring the Power of Tables on the Web [12] . . . 181
8.3.1 Relations . . . 182
8.3.2 Relation Search . . . 182
8.3.3 Indexing . . . 183
8.3.4 ACSDb Applications . . . 183

9 Materialized Views, Cubes and Aggregation 185
9.1 Materialized Views [14] . . . 185
9.2 On the Computation of Multidimensional Aggregates [29] . . . 185
9.2.1 Optimizations Possible . . . 186
9.2.2 Multiple Independent Group-By Queries (Independent Method) . . . 186
9.2.3 Hierarchy of Group-By Queries (Parent Method) . . . 186
9.2.4 Overlap Method . . . 186
9.2.5 Some Important Issues . . . 187
9.3 Implementing Data Cubes Efficiently [30] . . . 187
9.3.1 The Lattice Framework . . . 188
9.3.2 The Cost Model . . . 189
9.3.3 Optimizing Data-Cube Lattices . . . 189
9.3.4 The Hypercube Lattice . . . 189
9.4 Informix Under Control: Online Query Processing [32] . . . 190
9.4.1 Application scenarios and performance requirements . . . 190
9.4.2 Randomized Data Access and Physical Database Design . . . 191
9.4.3 Preferential data delivery: Online reordering . . . 191
9.5 BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data [2] . . . 191

9.5.1 Background . . . 192
9.5.2 System Overview . . . 193
9.5.3 Sample Creation . . . 193
9.5.4 BlinkDB Runtime . . . 194

10 Special-case Data Models: Streams, Semistructured, Graphs 195

10.1 The CQL Continuous Query Language: Semantic Foundations and Query Execution [5] . . . 195
10.1.1 Introduction to Running Example . . . 196
10.1.2 Streams and Relations . . . 196
10.1.3 Abstract Semantics . . . 196
10.1.4 Linear Road in CQL . . . 197
10.1.5 Time Management . . . 198
10.1.6 Implementation . . . 198
10.2 Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases [21] . . . 200
10.2.1 DataGuides . . . 200
10.2.2 Existence of Multiple DataGuides . . . 200
10.3 PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs [22]202
10.3.1 Related Work . . . 202
10.3.2 PowerGraph Abstraction . . . 203
10.3.3 Other Details . . . 204
10.3.4 Distributed Graph Placement . . . 204

11 Data Integration, Provenance and Transformation 205

11.1 Schema Mapping as Query Discovery [34] . . . 205
11.1.1 Value Correspondences . . . 205

11.1.2 Constructing Schema Mappings . . . 205
11.1.3 Query Discovery Algorithm . . . 206
11.1.4 Provenance in Databases: Why, How, and Where (Cheney et al.) . . . 206
11.1.5 Why-provenance . . . 206
11.1.6 How-Provenance . . . 207
11.1.7 Where-Provenance . . . 207
11.1.8 Eager vs Lazy . . . 207
11.2 Wrangler: Interactive Visual Specification of Data Transformation Scripts [35] 207
11.2.1 Wrangler Interface Design . . . 208
11.2.2 Wrangler Inference Engine . . . 209

12 Systems Support for ML **210**

12.1 The MADlib Analytics Library [31] . . . 210
12.1.1 Macro Programming . . . 211
12.1.2 Micro-Programming . . . 211
12.2 HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent [46] . . . 212
12.2.1 Sparse Separable Cost Functions . . . 212
12.2.2 The HOGWILD! Algorithm . . . 213
12.2.3 Fast Rates for Lock-Free Parallelism . . . 213
12.3 Scaling Distributed Machine Learning with the Parameter Server [38] . . . 213
12.3.1 Architecture . . . 214
12.3.2 Implementation . . . 215

1 Basics and Foundational Systems

In this section, we discuss the following papers:

- Architecture of a Database System
- The Five Minute Rule Twenty Years Later
- A History and Evaluation of System R
- The POSTGRES Next-Generation Database System
- The Gamma Database Machine Project

1.1 Architecture of a Database System [33]

Abstract

Database Management Systems (DBMSs) are a ubiquitous and critical component of modern computing, and the result of decades of research and development in both academia and industry. Historically, DBMSs were among the earliest multi-user server systems to be developed, and thus pioneered many systems design techniques for scalability and reliability now in use in many other contexts. While many of the algorithms and abstractions used by a DBMS are textbook material, **there has been relatively sparse coverage in the literature of the systems design issues that make a DBMS work.** This paper presents an architectural discussion of DBMS design principles, including process models, parallel architecture, storage system design, transaction system implementation, query processor and optimizer architectures, and typical shared components and utilities. Successful commercial and open-source systems are used as points of reference, particularly when multiple alternative designs have been adopted by different groups.

Consider an application where a gate agent at an airport requests the list of passengers for a flight. The following is the life of a query, in a nutshell (see Figure 1):

1. The client establishes a connection with the Client Communications Manager of a DBMS. In some cases, this is established directly via the ODBC or JDBC connectivity protocol (also known as a two-tier solution). In other cases, the client may communicate with a "middle-tier server" (a web server, transaction processing monitor, or the like), which in turn uses a protocol to proxy the communication between the client and the DBMS. This is also known as a three-tier solution.

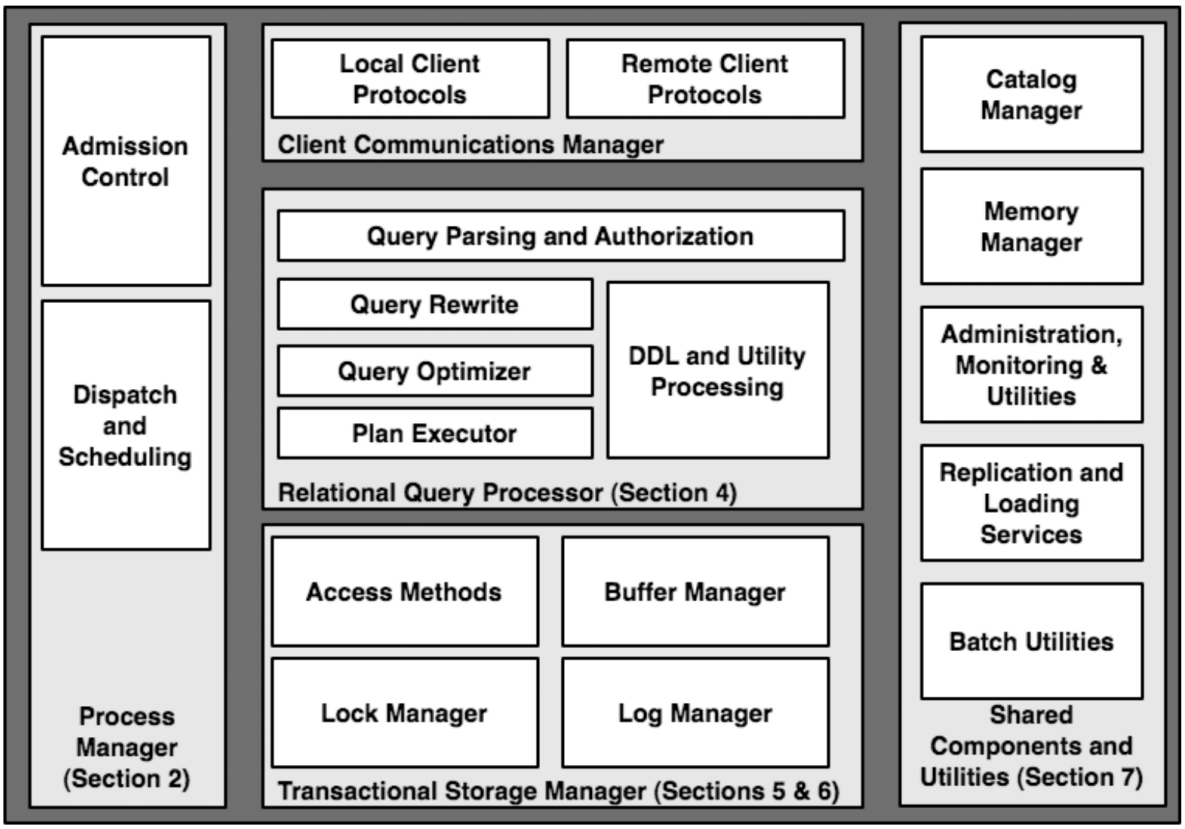

Figure 1: Figure 1 from Architecture of a Database System.

2. Upon receiving the client's first SQL command, the DBMS assigns a "thread of computation" to the command via its Process Manager. The most important decision that the DBMS needs to make at this stage in the query regards admission control: whether the system should begin processing the query immediately, or defer execution until a time when enough system resources are available to devote to this query.

3. The query executes by invoking the code in the Relational Query Processor. This set of modules checks that the user has permissions, compiles a query plan, and gives it to the plan executor. The leaves of the query plan are "access methods" for the storage layer.

4. For each operator in the query plan, operators make calls to fetch data from the DBMS' Transactional Storage Manager, which manages all data access (read) and manipulation (create, update, delete) calls. The storage system includes algorithms and data structures (e.g., tables, indexes) and a buffer management module that decides when and what data to transfer between disk and memory buffers. Locks are also acquired and managed in this module.

5. The stack of activities up till this point is unwinded, and state is freed by each of the modules.

1.1.1 Process Models

There are a variety of process models. Different decisions make sense these days depending on your platform. This discussion exposes historical mismatches/distrust across OS and DBMS groups/vendors. Here are some definitions we'll use:

1. OS process: combines an operating system (OS) program execution unit (a thread of control) with an address space private to the process. Included in the state maintained for a process are OS resource handles and the security context. This single unit of program execution is scheduled by the OS kernel and each process has its own unique address space.

2. OS thread: OS program execution unit without additional private OS context and without a private address space. Each OS thread has full access to the memory of other threads executing within the same multithreaded OS Process. Thread execution is scheduled by the operating system kernel scheduler and these threads are often called "kernel threads" or k-threads.

3. Lightweight thread package: application-level construct that supports multiple threads within a single OS process. Unlike OS threads scheduled by the OS, lightweight threads are scheduled by an application-level thread scheduler. The difference between a lightweight thread and a kernel thread is that a lightweight thread is scheduled in user-space without kernel scheduler involvement or knowledge.

4. DBMS worker: the thread of execution in the DBMS that does work on behalf of a DBMS Client. A 1:1 mapping exists between a DBMS worker and a DBMS Client: the DBMS worker handles all SQL requests from a single DBMS Client.

Lightweight threads have advantages and disadvantages. An advantage is that there is no need for OS kernel mode switch to schedule the next thread. A disadvantage is that blocking I/O requests will block all the other threads in the process, so lightweight thread packages must only issue non-blocking I/O requests. This makes for a more difficult programming model.

In the context of working on a single processor, single machine system, the DBMS has 3 possible process models, ranging from simplest to most complex: (1) process per DBMS worker, (2) thread per DBMS worker, and (3) process pool.

Process per DBMS worker. DBMS workers are mapped directly onto OS processes. The OS scheduler manages the timesharing of DBMS workers and the DBMS programmer can rely on OS protection facilities to isolate standard bugs like memory overruns. The scaling issues arise because a process has more state than a thread and consequently consumes more memory. A process switch requires switching security context, memory manager state, file and network handle tables, and other process context. Thread switches don't require this. This model is supported by IBM DB2, PostgreSQL, and Oracle.

Thread per DBMS worker. A single multithreaded process hosts all the DBMS worker activity. A dispatcher thread listens for new DBMS client connections, and each connection is allocated a new thread. Challenges are: locking / making sure threads access shared memory correctly, handling race conditions, and porting across different OSes. This model scales much better to a large number of concurrent connections though and is used by IBM DB2, Microsoft SQL Server, MySQL, Informix, and Sybase. IBM DB2 and MySQL use OS thread per DBMS worker.

Process pool. This model is a variant of process per DBMS worker. Rather than allocating a full process per DBMS worker, they are hosted by a pool of processes. A central process holds all DBMS client connections and, as each SQL request comes in from a client, the request is given to one of the processes in the process pool. SQL Server uses this model for 99% of use cases.

In the thread per DBMS worker model, data sharing is easy with all threads run in the same address space. In other models, shared memory is used for shared data structures and state. Additionally, we need ways to move data from the server back to the client. The two major types are disk I/O buffers and client communication buffers.

Disk I/O buffers. All persistent database data is staged through the DBMS buffer pool. With thread per DBMS worker, the buffer pool is simply a heap-resident data structure available to all threads in the shared DBMS address space. In the other two models, the

buffer pool is allocated in shared memory available to all processes. For the log tail—the log tail is simply a heap-resident data structure for the thread per DBMS worker. For the other two models, we might use a separate process to manage the log. Log records are communicated to the log manager by shared memory or any other efficient inter-process communications protocol.

Client communication buffers. Clients consume result tuples from a query cursor by repeatedly issuing the SQL `FETCH` request, which retrieve one or more tuples per request. Most DBMSs try to work ahead of the stream of `FETCH` requests to enqueue results in advance of client requests.

Finally, we will discuss admission control. As the workload in any multi-user system increases, throughput will increase up to some maximum. Beyond this point, it will begin to decrease radically as the system starts to thrash. Thrashing means that the connections need more working memory than what our buffer is capable of, so our DBMS spends most of the time just replacing pages. Thrashing can also be a result of lots of deadlock.

Definition 1.1. Any good DBMS will have an admission control policy, which does not accept new work unless sufficient DBMS resources are available.

There are 2 tiers of admission control. The first one—a simple policy—may be in the dispatcher process to ensure that the number of client connections is kept below a threshold. The second layer of admission control must be implemented directly within the core DBMS relational query processor. Why? To control how much memory is being requested & needed to be used during operations like sorts and joins.

1.1.2 Parallel Architecture

A shared-memory parallel system is one in which all processors can access the same RAM and disk with roughly the same performance. This architecture is fairly standard today — most server hardware ships with between two and eight processors. The main challenge is to modify the query execution layers to take advantage of the ability to parallelize a single query across multiple CPUs. There are ”shared-nothing” and ”shared-disk” models:

Shared-nothing system. In this cluster of independent machines, there is no way for a given system to directly access the memory or disk of another system. Coordination of the various machines entirely in the hands of the DBMS. The most common technique employed by DBMSs to support these clusters is to run their standard process model on each machine, or node, in the cluster. The main difference is that each system in the cluster

stores only a portion of the data—tables are spread over multiple systems in the cluster using horizontal data partitioning to allow each processor to execute independently of the others. During query execution, the query optimizer chooses how to horizontally re-partition tables and intermediate results. The DB administrator, as a result, has a big burden of partitioning tables appropriately to get good performance. Another challenge is that explicit cross-processor coordination must take place to handle transaction completion, provide load balancing, and support certain maintenance tasks. For example, the processors must exchange explicit control messages for issues like distributed deadlock detection and two-phase commit. Finally, what do you do for partial failure? One option is to bring down all nodes if one node fails. Another option is Informix's "data skip" approach—it just skips the data in the failed node. Another option is to employ some data redundancy approach. In chained declustering, when a node does fail, the system load is distributed fairly evenly over the remaining nodes: the $n - 1$ remaining nodes each do $n/(n - 1)$ of the original work, and this form of linear degradation in performance continues as nodes fail. Shared-nothing is most commonly used today, especially in data warehouses.

Shared-disk system. All processors can access the disks with about the same performance, but are unable to access each other's RAM. Oracle and IBM DB2 use this. One potential advantage of shared-disk over shared-nothing systems is their lower cost of administration: DBAs of shared-disk systems do not have to consider partitioning tables across machines in order to achieve parallelism. However, in massive databases, this is a moot point as partitioning is still required. Another advantage is that a single node's failure is not terribly complicated to handle. Failure typically happens when the data is corrupt—it can still be redundantly stored, for example. A disadvantage of this model is that explicit coordination of data sharing across the machines is needed. It needs a cache-coherency protocol for managing the distributed buffer pools.

Non-Uniform Memory Access (NUMA) systems. Each system in the cluster can access its own local memory quickly, whereas remote memory access across the high-speed cluster interconnect is somewhat delayed. NUMA hardware architectures are an interesting middle ground between shared-nothing and shared-memory systems. They are much easier to program than shared-nothing clusters, and also scale to more processors than shared-memory systems by avoiding shared points of contention such as shared-memory buses. They haven't been too successful, except in shared memory multi-processors. When the ratio of nearmemory to far-memory access times rises above the 1.5:1 to 2:1 range, the DBMS needs to employ optimizations to avoid serious memory access bottlenecks.

One potential problem that arises from implementing thread per DBMS worker using DBMS threads becomes immediately apparent when using multiprocessor hardware. When mapping DBMS threads to multiple OS processes, decisions need to be made about how many OS processes to employ, how to allocate the DBMS threads to OS threads, and how to distribute across multiple OS processes. A good rule of thumb is to have one process

per physical processor.

1.1.3 Relational Query Processor

A relational query processor takes a declarative SQL statement, validates it, optimizes it into a procedural dataflow execution plan, and (subject to admission control) executes that dataflow program on behalf of a client program. The client program then fetches ("pulls") the result tuples, typically one at a time or in small batches. In general, relational query processing can be viewed as a single-user, single-threaded task. Concurrency control is managed transparently by lower layers of the system.

In this section we focus on the common-case SQL commands: Data Manipulation Language (DML) statements including SELECT, `INSERT`, `UPDATE`, and `DELETE`. Data Definition Language (DDL) statements such as `CREATE TABLE` and `CREATE INDEX` are typically not processed by the query optimizer.

1. **Query Parsing:** the main tasks for the SQL Parser are to (1) check that the query is correctly specified, (2) resolve names and references, (3) convert the query into the internal format used by the optimizer, and (4) verify that the user is authorized to execute the query. . It canonicalizes table names into a fully qualified name of the form server.database.schema.table (four-part name). After canonicalizing the table names, the query processor then invokes the catalog manager to check that the table is registered in the system catalog and make sure the attribute references are correct. It is possible to constraint-check constant expressions during compilation as well; however, deferring this to execution time is quite common.

2. **Query rewrite:** simplifies and normalizes the query without changing its semantics. It can rely only on the query and on metadata in the catalog, and cannot access data in the tables. The rewriter's main responsibilities are: view expansion, constant arithmetic evaluation, logical rewriting of predicates (including adding transitive predicates), semantic optimization (e.g., redundant join elimination), and subquery flattening/other heuristic rewrites.

3. **Query optimizer:** transforms an internal query representation into an efficient query plan for executing the query. The original System R prototype compiled query plans into machine code, whereas the early INGRES prototype generated an interpretable query plan. There are multiple components to query optimization: plan space (e.g., only consider left-deep plans, where the right-hand input to a join must be a base table and postpone Cartesian joins), selectivity estimation (e.g. histograms), search algorithms (e.g., top-down cascades), parallelization (e.g., two phase optimization: optimize then distribute), and auto-tuning.

Note 1.1. SQL supports the ability to "prepare" a query: to pass it through the parser, rewriter and, optimizer, store the resulting query execution plan, and use it in subsequent "execute" statements. This is good except when the distribution of the data changes, or the values in the query statement (e.g., in a filter clause) are now atypical. Query preparation is especially useful for form-driven, canned queries over fairly predictable data.

As a database changes over time, it often becomes necessary to reoptimize prepared plans. Some vendors (e.g., IBM) work very hard to provide predictable performance across invocations at the expense of optimal performance per invocation. As a result, they will not re-optimize a plan unless it will no longer execute, as in the case of dropped indexes. Other vendors (e.g., Microsoft) work very hard to make their systems self-tuning, and will re-optimize plans more aggressively—e.g., if the cardinality of a table changes significantly, recompilation will be triggered in SQL Server since this change may influence the optimal use of indexes and join orders.

4. **Query Executor:** Most modern query executors employ the iterator model that was used in the earliest relational systems. This is easy to port to multiprocessor models. The basic iterator superclass logic never allocates memory dynamically, raising the question: where are the actual tuples being referenced stored in memory? It can be in the buffer pool (BP-tuples) or memory heap (M-tuples). Exclusive use of M-tuples can be a major performance problem, since memory copies are often a serious bottleneck in high-performance systems. The most efficient approach is to support tuple descriptors that can reference both BP-tuples and M-tuples. One challenge in this step is that the mix of reading and writing the same table (possibly multiple times) requires some care.

Definition 1.2. The Halloween Problem—the System R developers discovered that some employees would get successive raises with a statement like: `UPDATE EMP; SET salary=salary*1.1; WHERE salary < 20000`. In the naive plan, the employees would receive raises until their salary was more than 20k. The index scan modifies a tuple and then moves it rightward in the tree, susceptible to being "rediscovered." The more correct solution is to first identify all the tuples that should be updated, then do the update. SQL semantics forbid this behavior: a single SQL statement is not allowed to "see" its own updates.

5. **Access Methods:** are the routines that manage access to the various disk-based data structures that the system supports (includes unordered files/heaps and indexes). In the init of an access method, you pass in SARGS (search arguments). There are two reasons to pass SARGs into the access method layer. The first reason should be clear: index access methods like B+-trees require SARGs in order to function

efficiently. The second reason is a more subtle performance issue: logic to check the SARG might need to pin/unpin or copy/delete tuples when the SARG isn't satisfied—so it's less wasteful to have the access method check the validity of the SARG. SARGS keep a nice clean architectural boundary between the storage engine and the relational engine while obtaining excellent CPU performance. Additionally, rows are "pointed to" via a row ID—a separate index is used to store a mapping from row ID to the actual, physical row ID. Only using physical RIDs sucks—when a physical RID changes (e.g., rebalancing a B+ tree), all indexes that have it must reupdate.

Now, in the era of data warehouses, the conventional query optimization and execution engines discussed so far do not work so well. Some optimizations include:

- Bitmap indexes: Data warehouses often have columns with a small number of values. A traditional B+ tree index will require (value, record-pointer) pairs for each record. Instead, we can use a bitmap and store just the values.
- Fast load: It is crucial that data warehouses be bulk-loadable very quickly. Although one could program warehouse loads with a sequence of SQL insert statements, this tactic is never used in practice. Instead a bulk loader is utilized that will stream large numbers of records into storage without the overhead of the SQL layer, and taking advantage of special bulk-load methods for access methods like B+-trees. If one keeps before and after values of updates, suitably timestamped, then one can provide queries as of a time in the recent past. Running a collection of queries as of the same historical time will provide compatible answers. Moreover, the same historical queries can be run without setting read locks.
- Materialized Views: Data warehouses are typically gigantic, and queries that join multiple large tables have a tendency to run "forever." To speed up performance on popular queries, most vendors offer materialized views. There are three aspects to Materialized View use: (a) selecting the views to materialize, (b) maintaining the freshness of the views, and (c) considering the use of materialized views in ad-hoc queries.
- OLAP and Ad-hoc Query Support: predictable queries can be supported by appropriately constructed materialized views. Since most business analytics queries ask for aggregates, one can compute a materialized view which is the aggregate value by some key: these aggregates are called data cubes.
- Optimization of Snowflake Schema Queries: many data warehouses follow a particular method for schema design: the star schema (fact and dimension). A central fact table surrounded by dimensions, each with a 1-N primary-key-foreign-key relationship to the fact table.

Finally, the relational query processor supports extensibility. Some extensions include:

- Abstract data types: A relational DBMS can be made extensible to new abstract data types at runtime, as was illustrated in the early IngresADT system, and more aggressively in the follow-on Postgres system. To achieve this, the DBMS type system—and hence the parser—has to be driven from the system catalog, which maintains the list of types known to the system, and pointers to the "methods" (code) used to manipulate the types. The DBMS does not interpret types, it merely invokes their methods appropriately in expression evaluation. The query optimizer has to account for "expensive" user-defined code in selection and join predicates, and in some cases postpone selections until after joins. To make ADTs even more efficient, it is useful to be able to define indexes on them.

- Structured Types and XML: There have been many proposals for more aggressive changes to databases to support non-relational structured types: i.e., nested collection types like arrays, sets, trees, and nested tuples and/or relations. There are roughly three approaches to handling structured types like XML. The first is to build a custom database system that operates on data with structured types. The second approach is to treat the complex type as an ADT. A third approach is for the DBMS to "normalize" the nested structure into a set of relations upon insertion, with foreign keys connecting subobjects to their parents (also known as "shredding"). Shredding adds storage overhead, and requires joins to "reconnect" the data upon querying.

- Full-text search: The common belief is that a naive implementation of text search in a DBMS like the one above runs roughly an order of magnitude slower than the custom text indexing engines. In most cases the full-text index is updated asynchronously ("crawled") rather than being maintained transactionally; PostgreSQL is unusual in offering the option of a full-text index with transactional updates. A key challenge in handling full-text search in a relational database is to bridge the semantics of relational queries (unordered and complete sets of results) with ranked document search using keywords (ordered and typically incomplete results) in a way that is useful and flexible.

- Additional issues: extensible query optimizers and remote data sources.

1.1.4 Storage Management

Two basic types of DBMS storage managers are in commercial use today: either (1) the DBMS interacts directly with the low-level blockmode device drivers for the disks (often called raw-mode access), or (2) the DBMS uses standard OS file system facilities. This decision affects the DBMS's ability to control storage in both space and time.

Spatial Control. Sequential bandwidth to and from disk is between 10 and 100 times faster than random access, and this ratio is increasing. As a result, it is critical for the DBMS storage manager to place blocks on the disk such that queries that require large amounts of data can access it sequentially. Since the DBMS can understand its workload

access patterns more deeply than the underlying OS, it makes sense for DBMS architects to exercise full control over the spatial positioning of database blocks on disk. The best way for the DBMS to control spatial locality of its data is to store the data directly to the "raw" disk device and avoid the file system entirely. The drawbacks are: (1) it requires the DBA to devote entire disk partitions to the DBMS, which makes them unavailable to utilities (backups, etc.) that need a filesystem interface. Second, "raw disk" access interfaces are often OSspecific, which can make the DBMS more difficult to port. Finally, we are now at a point where "virtual" disk devices are the norm in most scenarios today — the "raw" device interface is actually being intercepted by appliances or software that reposition data aggressively across one or more physical disks. As a result, the benefits of explicit physical control by the DBMS have been diluted over time. An alternative to raw disk access is for the DBMS to create a very large file in the OS file system, and to manage positioning of data as offsets in that file.

Temporal Control: Buffering. In addition to controlling where on the disk data should be placed, a DBMS must control when data gets physically written to the disk. Most OS file systems also provide built-in I/O buffering mechanisms to decide when to do reads and writes of file blocks. If the DBMS uses standard file system interfaces for writing, the OS buffering can confound the intention of the DBMS logic by silently postponing or reordering writes. We care about correctness—we need to flush in the right order at the right time. Given that the DBMS has to do its own buffering carefully for correctness, any additional buffering by the OS is redundant. `mmap` can help with double-buffering. Additionally, the DBMS is better at speculative reads and delayed writes.

Buffer Management. The buffer pool is organized as an array of frames, where each frame is a region of memory the size of a database disk block. Associated with the array of buffer pool frames is a hash table that maps (1) page numbers currently held in memory to their location in the frame table, (2) the location for that page on backing disk storage, and (3) some metadata about the page. The metadata includes a dirty bit to indicate whether the page has changed since it was read from disk, and any information needed by the page replacement policy to choose pages to evict when the buffer pool is full. Most systems also include a pin count to signal that the page is not eligible for participation in the page-replacement algorithm. What policies are used to determine page replacement? The DBMS has a variety of workloads that would benefit from different policies. For example, certain database operations tend to require full table scans and, when the scanned table is much larger than the buffer pool, these operations tend to clear the pool of all commonly referenced data—so recency of reference is a poor predictor of the probability of future reference, invalidating schemes like LRU and CLOCK. Today, most systems use simple enhancements to LRU schemes to account for the case of full table scans. Another strategy used in commercial systems is to have the replacement policy depend on the page type: e.g., the root of a B+-tree might be replaced with a different strategy than a page in a heap file. Recent hardware trends, including 64-bit addressing and falling memory prices, have made very large buffer pools economically possible.

Database storage subsystems are a very mature technology, but a number of new considerations have emerged for database storage in recent years, which have the potential to change data management techniques in a number of ways. One key technological change is the emergence of flash memory as an economically viable random-access persistent storage technology. Additionally, as processor performance has improved and RAM latencies have not kept pace, it has become increasingly important to consider keeping data compressed even during computations, so as to maximize the residency of data in processor caches as well. Finally, outside the traditional relational database market there is enhanced interest in large-scale but sparse data storage techniques, where there are logically thousands of columns, the bulk of which are null for any given row. This motivates column-oriented storage.

1.1.5 Transactions: Concurrency Control and Recovery

The truly monolithic piece of a DBMS is the transactional storage manager that typically encompasses four deeply intertwined components:

1. A lock manager for concurrency control.
2. A log manager for recovery.
3. A buffer pool for staging database I/Os.
4. Access methods for organizing data on disk.

Of note are the ACID principles. ACID stands for Atomicity, Consistency, Isolation, and Durability. These terms were not formally defined, and are not mathematical axioms that combine to guarantee transactional consistency.

- Atomicity: "all or nothing" guarantee
- Consistency: application-specific guarantee; SQL integrity constraints are typically used to capture these guarantees in a DBMS
- Isolation: guarantee to application writers that two concurrent transactions will not see each other's in-flight (notyet-committed) updates
- Durability: e updates of a committed transaction will be visible in the database to subsequent transactions independent of subsequent hardware or software errors

Durability is typically implemented via logging and recovery. Isolation and Atomicity are guaranteed by a combination of locking (to prevent visibility of transient database states), and logging (to ensure correctness of data that is visible). Consistency is managed

by runtime checks in the query executor: if a transaction's actions will violate a SQL integrity constraint, the transaction is aborted and an error code returned.

Serializability. This principle dictates that a sequence of interleaved actions for multiple committing transactions must correspond to some serial execution of the transactions—as though there were no parallel execution at all. Serializability is a way of describing the desired behavior of a set of transactions. Isolation is the same idea from the point of view of a single transaction. There are three broad techniques of concurrency control enforcement.

1. In strict 2 phase locking (2PL), transactions acquire a shared lock on every data record before reading it, and an exclusive lock on every data item before writing it. All locks are held until the end of the transaction, at which time they are all released atomically.
2. Multi-version concurrency control (MVCC): Transactions do not hold locks, but instead are guaranteed a consistent view of the database state at some time in the past, even if rows have changed since that fixed point in time.
3. In Optimistic Concurrency Control (OCC), multiple transactions are allowed to read and update an item without blocking. Transactions maintain histories of their reads and writes. Before committing a transaction checks history for isolation conflicts they may have occurred; if any are found, one of the conflicting transactions is rolled back.

Most commercial relational DBMS implement full serializability via 2PL. The lock manager is the code module responsible for providing the facilities for 2PL.

Locking. Every lock is associated with a transaction and each transaction has a unique transaction ID. Locks come in different lock "modes," and these modes are associated with a lock-mode compatibility table. Hierarchical locking allows a single lock to be used to lock an entire table and, at the same time, support row granularity locks in the same table both efficiently and correctly. A global lock table is maintained to hold lock names and their associated information. The lock table is a dynamic hash table keyed by (a hash function of) lock names. Associated with each lock is a mode flag to indicate the lock mode, and a wait queue of lock request pairs (transactionID, mode). In addition, the lock manager maintains a transaction table keyed by transactionID, which contains two items for each transaction T: (1) a pointer to T's DBMS thread state, to allow T's DBMS thread to be rescheduled when it acquires any locks it is waiting on, and (2) a list of pointers to all of T's lock requests in the lock table, to facilitate the removal of all locks associated with a particular transaction (e.g., upon transaction commit or abort).

Latching. As an auxiliary to database locks, lighter-weight latches are also provided for mutual exclusion. As an example, the buffer pool page table has a latch associated with

each frame, to guarantee that only one DBMS thread is replacing a given frame at any time. Latches are implemented using an atomic hardware instruction. The lock manager tracks all the locks held by a transaction and automatically releases the locks in case the transaction throws an exception, but internal DBMS routines that manipulate latches must carefully track them and include manual cleanup as part of their exception handling. Latches differ from locks in a number of ways:

- Locks are kept in the lock table and located via hash tables; latches reside in memory near the resources they protect, and are accessed via direct addressing.
- Locks are typically subject to the strict 2PL protocol; latches are not
- Lock acquisition is largely in the hands of applications and the query optimizer. Latches are acquired by specialized code inside the DBMS, and the DBMS internal code issues latch requests and releases strategically.
- Locks are allowed to produce deadlock, and lock deadlocks are detected and resolved via transactional restart. Latch deadlock must be avoided; the occurrence of a latch deadlock represents a bug in the DBMS code
- Latch calls take at most a few dozen CPU cycles whereas lock requests take hundreds of CPU cycles
- Latches are not tracked and so cannot be automatically released if the task faults

Transaction Isolation Levels.

[Shreya: TODO]

Log Manager.

[Shreya: TODO]

Locking and Logging in Indexes.

[Shreya: TODO]

1.1.6 Shared Components

[Shreya: TODO]

1.2 The Five Minute Rule Twenty Years Later [25]

In 1987, Jim Gray and Gianfranco Putzolu published their now-famous five-minute rule for trading off memory and I/O capacity. They found that the rice of RAM to hold a 1KB record was about equal to the fractional price of a disk drive required to access such a record every 400 seconds (rounded to 5 minutes). The break-even interval is inversely proportional to the record size (the larger the record, the smaller the access interval should be to warrant putting the record in RAM).

How is this 5-minute rule derived? Suppose we have page size $PAGESIZE$, access interval $ACCESSINTERVAL$ (in seconds), $RAMCOST$ (dollars per byte), $ACCESSCOST$ (dollars per page per second), and a record with $RECORDSIZE$ bytes. The cost of storing in RAM is simply $STORAGECOST_{RAM} = RAMCOST \times RECORDSIZE$. The cost of storing on disk is:

$$STORAGECOST_{DISK} = \left\lceil \frac{RECORDSIZE}{PAGESIZE} \right\rceil \times \frac{ACCESSCOST}{ACCESSINTERVAL}$$

Now, we are interested in knowing the break-even interval, or the $ACCESSINTERVAL$ required to merit storing the object in RAM. Solving the inequality, we get:

$$\begin{aligned} STORAGECOST_{RAM} &\leq STORAGECOST_{DISK} \\ RAMCOST \times RECORDSIZE &\leq \left\lceil \frac{RECORDSIZE}{PAGESIZE} \right\rceil \times \frac{ACCESSCOST}{ACCESSINTERVAL} \\ ACCESSINTERVAL &\leq \left\lceil \frac{RECORDSIZE}{PAGESIZE} \right\rceil \times \frac{ACCESSCOST}{RAMCOST \times RECORDSIZE} \end{aligned}$$

This paper asks the question: how does flash memory influence the 5 minute rule? The paper says that flash sits between RAM and disk and says that it can be viewed either as extended RAM or extended disk. It argues that file systems should use it as extended RAM and that databases should use it as extended disk. If we compute new five minute rules for RAM/disk, RAM/flash, and flash/disk for 4 KB objects, we get 1.5 hours, 15 minutes, and 2.5 hours. Why should the filesystem use flash as extended RAM? Well, if it view flash as disk, it performs metadata-related writes to flash and eventually the writes are moved to disk. Thus, we have to double the amount of persistent writes. A database simply logs stuff in a log, so it doesn't have to do many persistent writes, so treating flash as disk is ok.

Every so often, a database has to checkpoint stuff somewhere persistent. Using flash as a disk lets it write checkpoints faster. An OS doesn't checkpoint, so it should treat flash as RAM. With a disk, seek time dominates transfer time, so it's better to have bigger B+ tree

nodes. With flash, seek time is less compared to transfer time, so we can shrink our B+ tree nodes a bit.

With regards to query processing, unclustered index joins might become faster than full table scans since the flash seek time is much lower.

1.3 A History and Evaluation of System R [13]

Summary

System R, an experimental database system, was constructed to demonstrate that the usability advantages of the relational data model can be realized in a system with the complete function and high performance required for everyday production use. This paper describes the three principal phases of the System R project and discusses some of the lessons learned from System R about the design of relational systems and database systems in general.

System R is essentially the first of relational databases and introduced influential ideas in all parts of a DBMS. Key goals of System R were:

1. To provide a high-level, nonnavigational user interface for user productivity and data independence
2. To support different types of DB use including transactions, ad hoc queries, and report generation
3. To support a rapidly changing DB environment, in which tables, indexes, views, transactions, and other objects could easily be added to and removed from the DB
4. To support concurrent users
5. To provide recovery
6. To provide the ability to define different views of data and give different permissions to different users
7. To have good performance for all of the above

The development was divided into 3 phases. Phase 0 was an experimental phase that primarily honed SQL as a language. Phase 1 was the first version of System R being truly built. Phase 2 involved the evaluation of System R.

1.3.1 Phase Zero

From the beginning, the intention was to learn what they could from the initial prototype, and then scrap the Phase Zero code before constructing System R. There were several key learnings:

- Query language: they did not support joins, but they learned joins were important.
- Optimizer: they tried to minimize the number of tuples read. Instead, they should have minimized I/O count and CPU cost. It would have revealed the importance of clustering together related tuples on physical pages.
- Access methods and physical storage: tuples stored pointers into "domains" where the data was actually stored. Inversions were maps from domain back to tuple ID. This meant reading took multiple IOs. Inversions were used to find tuples containing a particular value. TID lists were large and stored as temporary objects in the database during query processing. A system catalog also existed.
- Concurrency and recovery: none

1.3.2 Phase One

System R consisted of an access method (called RSS, the research storage system) and optimizing SQL processor (called RDS, the relational data system). Separation of RSS and RDS was useful—e.g., all locking and logging were isolated in the RSS, while authorization and access path selection were isolated in the RDS.

This prototype supported multiple users, motivating a locking subsystem, view, and authorization subsystems. The standalone query interface of System R (called UFI, the user friendly interface) was supported by a dialog manager program. Some of the key learnings from this phase include:

- Query language: support joins
- Compilation: Queries should be compiled into compact, efficient routines (known as "fragments"). A preprocessor would examine, optimize, and compile SQL statements into an access module. All the overhead of parsing, validity checking, and access path selection was in a separate processor. System R recorded, with each access module, a list of "dependencies" on DB objects such as tables and indexes (provided by the separate preprocessor). When DB objects were modified or deleted, all dependency access modules are marked as invalid. If an invalid access module is invoked, it is recompiled.

- Access paths: Rather than storing data in separate domains, RSS stored data values in individual records in the DB. Tuples were stored row-by-row. This meant records were variable length, unfortunately. The advantage though was that all data values of a record could be fetched via single I/O. In place of inversions, the RSS provided indexes as B-trees. Access paths included index scans (i.e., going through values in order), relation scans (i.e., going through physical layout), and link scans (i.e., traversing from one record to another). Scans accepted search args (SARGs) to limit records to those satisfied by a particular predicate. System R primarily used index and relation scans for sorting. Links were used for internal purposes, not as an access path to user data.

- Optimizer: minimize a weighted combination of IOs and RSS calls (as a proxy for CPU cost). The optimizer scans each table in the query by means of one index (or if no index exists, a relation scan). The choice would be based on the optimizer's estimate of clustering and selectivity properties of each index, based on statistics stored in the system catalog. There were nested loop joins, index joins, and sort-merge joins.

- Recovery: used logging and shadow paging. As each page in the DB is updated, the page is written out in a new place on disk, and the original is retained. A directory of old and new locations of each page is maintained. Periodically they checkpointed—forced all updates out to disk, discard old pages, and make the new pages the old pages. To bring the DB back to a consistent state, the system reverts to the old pages and uses the log to redo committed transactions and undo updates made by incomplete transactions.

- Locking: the original design had the concept of predicate locks, but it was abandoned for multiple reasons. First, determining whether 2 predicates are mutually satisfiable is time-consuming. Second, two predicates might appear to conflict even when the semantics of the data prevent any conflicts. Third, they wanted the locking subsystem to be entirely within the RSS, without getting understanding about the predicates. They ended up using multigranular, hierarchical locks with intention locks. Each user holds an intention lock on a larger object and exclusive locks on particular records being updated.

- Authorization: views were primarily used for authorization.

1.3.3 Phase Two

The evaluation phase lasted about 2.5 years and consisted of two parts: experiments and case studies with actual users. User response was quite enthusiastic. It took a matter of days to install, populate, and put into application programs. Interactive response was bad though for complex SQL statements involving joins of several tables—causing them to question how useful normalization was.

Users consistently liked SQL syntax. They made a number of suggestions for extensions and improvements to the language:

1. Wanted syntax to easily check the existence of an item, motivating the `EXISTS` predicate
2. Wanted to search for character strings whose contents are only partially known, motivating `LIKE` predicate
3. Prepared statements—use the same statement for different values without reinvoking the optimizer, motivating `PREPARE` and `EXECUTE` statements
4. Outer join (was under study at time of publication)

Some other learnings include:

- Experiments showed that for a typical short transaction, 80% of instructions were executed by RSS, and remaining 20% were executed by the access module and application program. Thus, the user pays only a small cost for the flexibility of System R instead of directly programming with the RSS interface.
- Hashing and direct links might be useful for access paths on user data, particularly for "canned transactions" that only influence a few records
- In optimization, occasionally the optimizer's prediction was vastly different from the actual cost. This was for a variety of reasons, such as the optimizer's inability to predict how much data remains in the buffer during sorting.
- It would be helpful to provide authorization to "groups" of users.
- In recovery, shadow paging had the greatest impact on system performance. This is because data is constantly moving about (each updated page is written out to a new location on disk) and you need a directory to keep track of old and new versions of every page. Also, periodic checkpoints consume CPU time. Here they suggest ditching the shadow page concept in favor of a log of database updates (motivating write-ahead logging, probably).
- For transactions, there are 3 levels of isolation in system R. In level 1, you can read uncommitted data (but not update). In level 2, you cannot read uncommitted data, but you can end up reading data committed by another transaction in the middle of your transaction (nonrepeatable read). In level 3, you can get repeatable reads—you must acquire a lock on each record and hold it until the end of the transaction. They expected level 2 to be cheaper and level 3 safer, but they found level 3 had less CPU overhead because it was simpler to acquire and keep locks than acquire and release and rerequire. They noticed the convoy phenomenon, where when a process holds a high-traffic lock for too long, all the other processes queue up fast.

- Automatically maintaining a data catalog was well-liked by users.

Definition 1.3. The **convoy phenomenon** occurs when high-traffic locks are requested frequently and held for a short period of time. Example locks include the buffer pool and system log locks. If process P1 goes into a wait state while it holds a high-traffic lock, all other processes more or less immediately request this lock and form a long queue, which will never truly dequeue. System R got rid of the convoy problem by removing the FIFO requirement of granting locks in the queue—locks were given to a random process in the queue, hoping processes would finish up and not enter the queue again.

1.4 The POSTGRES Next-Generation Database System [49]

Abstract

Commercial relational Database Management Systems (DBMSs) are oriented toward efficient support for business data processing applications where large numbers of instances of fixed format records must be stored and accessed. The traditional transaction management and query facilities for this application area will be termed data management, and are addressed by relational systems. To satisfy the needs of users outside of business applications, DBMSs must be expanded to offer services in two other dimensions, namely object management and knowledge management. Object management entails efficiently storing and manipulating nontraditional data types such as bitmaps, icons, text, and polygons. Object management problems abound in CAD and many other engineering applications. Knowledge management entails the ability to store and enforce a collection of rules that are part of the semantics of an application. Such rules describe integrity constraints about the application, as well as allowing the derivation of data that is not directly stored in the database.

As relational databases shifted to commercialization, everybody in research wanted to move on to more stuff. Beyond business processing, new applications seemed to need more complex models (e.g., CAD, geospatial, manufacturing/supply chain, etc). Why are database engines doing trying to be code runtimes?! For data-centric code, "push code to data", rather than "pull data to code". This is a DB community chestnut, and the opposite anti-pattern recurs over and over. Also, it is surprising what you can and cannot express in Codd's relational languages: Cannot count, or even figure out even/odd polarity! A bunch of manifestos came out, thinking about what to do next.

POSTGRES philosophy. It's pitched as expanding into "data, object, and knowledge management." The query language as the primary interface (as opposed to imperative

OO programming) proved (and still proves) to be a win. The object model is simple: classes, inheritance, types, functions, and more. Much of this was folded into SQL over time, fairly naturally.

1.4.1 The POSTGRES Data Model and Query Language

The POSTGRES data model is relational, except that relations are now called classes and tuples are called instances. Also classes can inherit from other classes to inherit all their attributes. All objects are assigned a unique OID, and objects can contain links to one another (i.e. navigational).

There are three types of classes—real, virtual (i.e., views), and version (i.e., stored as a differential relative to a parent). There are three kinds of types in POSTGRES: base types; arrays of base types; and composite types.

There are three different kinds of functions known to POSTGRES: C functions; operators; and POSTQUEL functions. A user can define an arbitrary number of C functions whose arguments ;are base types or composite types. C functions can be defined to POSTGRES while the system is running and are dynamically loaded when required during query execution. UDFs can be chained to create UDAFs (user defined aggregate functions), which is useful to push close to the data. As an aside, MapReduce is the antithesis of this. To utilize indexes in processing queries, POSTGRES supports a second kind of function, called operators (e.g., custom greater than/less than). The third kind of function available in POSTGRES is POSTQUEL functions. Any collection of commands in the POSTQUEL query language can be packaged together and defined as a function.

Stonebraker remembers support for UDTs the most. If you register ordering operators for an ADT, B+ trees just work! DB catalog would keep track of a "template" for B-trees, and you would register UDFs to fill in the template for a new data type. And there are optimizer hooks—since optimizers need estimates on UDF runtimes and selectivity, you can register cardinality estimation formulas for each index operator. However, there are still unsolved problems in index extensibility—disk layouts, correct fine-grained concurrency control, and correct physical logging/recovery. Generalized Search Tree (GiST) provided a higher-level API that made index extensions more "user friendly" and correct for whatever search predicates you choose to support.

User code can directly call into POSTGRES internals via a fast path.

1.4.2 The Rules System

They tried to construct a general-purpose rules system that could perform all of the following functions: view management; triggers; integrity constraints; referential integrity;

protection; and version control. ECA: on Event if Condition do Action. The rules act like triggers: they are a set of instructions to run whenever a certain event (e.g. insertion, deletion, update, etc) is made. The rules can use forward or backwards chaining (e.g., when I write this, update that). Conflict resolution policies for rules are nontrivial—suppose 2 rules are triggered, which do you fire first and what happens if they trigger more rules? This body of work never overcame the issues that led to AI winter: the interactions within a pile of rules can become untenably confusing as the rule set grows even modestly

There are 2 implementations of POSTGRES rules. The first is through record-level processing, deep in the run-time system. Here, a marker is placed on attributes in the rule definitions, and if an executor touches a marked attribute, it calls the rule system before proceeding. The record-level system is good for a large number of rules that each cover only a few instances. The second is through a query rewrite module that exists between the parser and query optimizer; however, many rules lead to big queries. The two implementations have different semantics, up to the user to choose from. Also, a user has to choose whether to run a rule in the same transaction or a different transaction, and whether the rule should be fired immediately or after it is confirmed that the transaction has committed. The most compelling use case for trigger systems: materialized View maintenance. Triggers still exist in many DBMSs, but semantics of ECA rules are still a mess—and triggers are not that efficient.

Question 1.1. What are the 2 difference implementations of POSTGRES rules? Discuss strengths and weaknesses of each approach.

1.4.3 The Storage System

Recovery is complex. "When considering the POSTGRES storage system, we were guided by a missionary zeal to do something different." Their solution was to do a no-overwrite storage system (the DB is the log is the DB). This gives time travel for free, instantaneous recovery, and no crash recovery code. POSTGRES does not use logging; instead it is forced to write every single modification to disk. This makes it much slower. POSTGRES runs one process per user.

How do updates work in a no-overwrite system? The first version of a record is called the Anchor Point, which will have a chain of associated delta records on the same page. On an update or delete, the Xmax and Cmax (transaction ID of updater or deleter, command ID of updater or deleter) are updated for the Anchor Point. Then a delta record is created, with an Xmin and Cmin (transaction ID of updater or deleter, command ID of updater or deleter) and the ID of the old record.

POSTGRES did 2PL instead of MVCC because 2PL is shown to be superior in a conven-

tional main memory lock table setting. Timestamps are set at commit time. There are 3 levels of archiving: no archive, light archive, and heavy archive (i.e., need to access old versions regularly). For relations with no archive status, Tmin and Tmax are never filled in, since access to historical tuples is never required. Historical data can be forced to archive via the vacuum cleaner.

The performance was questionable—it was 2 times as fast as UCB ingres, but used fast path to implement all benchmark routines as C functions. NVRAM required to make POSTGRES compete on even this benchmark. Also, transactions had pretty strict assumptions—records fit on a single page, deltas and anchors were on the same page, transactions were single-record, and it was an "update-only" workload (so maybe a read-heavy workload might not have really benefitted from POSTGRES).

Question 1.2. Does POSTGRES use WAL for recovery? How does POSTGRES get recovery? Discuss the no-overwrite system used in Postgres.

They used a log of some sorts, but not a WAL. This log had 2 bits per transaction.

1.5 The Gamma Database Machine Project [19]

This paper describes the design of the Gamma database machine and the techniques employed in its implementation. Gamma is a relational database machine currently operating on an Intel iPSC/2 hypercube with 32 processors and 32 disk drives. Gamma employs three key technical ideas which enable the architecture to be scaled to 100s of processors. First, all relations are horizontally partitioned across multiple disk drives enabling relations to be scanned in parallel. Second, novel parallel algorithms based on hashing are used to implement the complex relational operators such as join and aggregate functions. Third, dataflow scheduling techniques are used to coordinate multioperator queries. By using these techniques it is possible to control the execution of very complex queries with minimal coordination - a necessity for configurations involving a very large number of processors. In addition to describing the design of the Gamma software, a thorough performance evaluation of the iPSC/2 hypercube version of Gamma is also presented.

In addition to measuring the effect of relation size and indices on the response time for selection, join, aggregation, and update queries, we also analyze the performance of Gamma relative to the number of processors employed when the sizes of the input relations are kept constant (speedup) and when the sizes of the input relations are increased proportionally to the number of processors (scaleup). The speedup results obtained for both selection and join queries are linear; thus, doubling the number of processors halves the response time for a query. The scaleup results obtained are also quite encouraging.

They reveal that a nearly constant response time can be maintained for both selection and join queries as the workload is increased by adding a proportional number of processors and disks.

Gamma is one of the first "massively parallel" (MPP) DBMSs. Concurrent with Teradata, Tandem, and Bubba, the canonical academic "shared nothing" parallel DBMS. Today, shared nothing is largely used for data warehousing workloads. But in the cloud, it's often the equivalent of shared disk, due to disaggregated storage and compute. Armando Fox has some interesting notes:

- Horizontal partitioning of relations across nodes (declustering) on user-selected index. In retrospect, this was a big mistake–should use "heat" of a relation (hot spot patterns?) to determine how to decluster it.
- Use hashes (specifically, a split table) to partition execution of query plan steps across nodes. Hash is applied (e.g.) to join attribute of tuples entering a join operation.
- Scheduler process can pipeline different phases of an operation by initiating them on different processors and rendezvousing the results. Example: to do a simple hash join of relation B with A, simultaneously initiate the build phase (in which tuples from inner relation A are inserted into a hash table; preparation for the probing phase, in which tuples from B are looked up in the hash table for matches) and the selection operator on A. When both complete, move on to the probing phase.
- Traditional intentional locking at file and page level; centralized deadlock detector; ARIES WAL and recovery.
- Chained declustering for availability:

Definition 1.4. In **chained declustering**, each disk holds the primary copy of tuple bucket k and the backup copy of bucket $k+1$ (mod the number of buckets). This is used for shared-nothing architectures. In contrast (shared-disk), in the RAID data storage scheme, an array of small, inexpensive disks is used as a single storage unit termed a group. Instead of replicating data on different drives, this strategy stores check (parity) bytes for recovering from both random single byte failures and single disk failures. If no disks have failed, reading one sector of a block requires access to only a single disk. Writing a sector, on the other hand, requires $n-1$ disk accesses: a read and write of the sector being updated plus a read and write of the check sector for the corresponding block. When a disk failure occurs, however, every disk in the same RAID group must be accessed each time an access is attempted to a sector on inoperative disk drive.

- Performance: Close to linear (but not perfectly linear) speedup and response latencies as processors are added for the measured workload. Disk and network overheads prevent performance from being perfectly linear. Also, for selections that end up selecting a small number of tuples, overhead to set up the selection dominates execution time and results in sublinear speedup.
- Flaw: Not surprising that a balanced system would require accounting for the high startup costs of disk operations. Aggregate disk bandwidth is not the problem: initiation and similar latencies are the problem. Surprising that the authors didn't deal with this up front.
- Flaw: As authors acknowledge, it's not necessarily always a good idea to horizontally split relations across disks (pessimal for some transactions).

The authors made 2 versions of Gamma to get a working system. V1 was on a cluster of VAXen. V2 was on an Intel hyprcube (still a shared-nothing software architecture). They wrote their own OS, NOSE, with lightweight processes (threads) and scheduling, a file system, semaphore services, and more. But they didn't have virtual memory.

In Gamma V1, their first big problem was that the token ring (how processors connected to each other) had a maximum network size of 2K bytes, and they set their page size to 2K bytes because they didn't want to deal with copying tuples from disk pages to network packets. However, they realized that the benefits of a larger disk page size more than offset the cost. The second problem was that the network interface on the processor was a huge bottleneck (4 megabits / second), which made the 80 megabits / second bandwidth of the token ring irrelevant. The transfer rate of pages from the disk was higher than the speed of the individual processors! Finally, since there was no virtual memory, they were limited to only 2MB per processor, which would be even less for multiprocessing. The problem was exacerbated by the fact that space for join hash tables, stack space for processes, and the buffer pool were managed separately in order to avoid flushing hot pages from the buffer pool. While there are advantages to having these spaces managed separately by the software, in a configuration where memory is already tight, balancing the sizes of these three pools of memory proved difficult. For transactions, they used 2PL, a deadlock detector, and ARIES. The biggest lesson learned: **a good high-bandwidth architecture is balanced: all the components can sustain the same throughput.**

The Gamma V2 took advantage of hardware with more memory and better networking. They had to port NOSE to the Intel hardware, to be a user-space thread package inside a UNIX process. Cute anecdote from the paper:

Note 1.2. "While we thought that making the necessary changes would be tedious but straightforward, we were about half way through the port before we realized that we would have to find and change every "for" loop in the system in which the loop index

was also used as the address of the machine to which a message was to be set. While this sounds silly now, it took us several weeks to find all the places that had to be changed. In retrospect, we should have made NOSE mask the differences between the two addressing schemes."

1.5.1 Software Architecture of Gamma

Gamma horizontally partitions all relations using round robin, hash, or range partitioning. Each node can maintain its own local indexes on its relations. At system initialization time, a UNIX daemon process for the Catalog Manager (CM) is initiated along with a set of Scheduler Processes, a set of Operator Processes, the Deadlock Detection Process, and the Recovery Process. The main components are:

- The function of the Catalog Manager is to act as a central repository of all conceptual and internal schema information for each database
- One query manager process is associated with each active Gamma user. The query manager is responsible for caching schema information locally, providing an interface for ad-hoc queries, query parsing, optimization, and compilation
- While executing, each multisite query is controlled by a scheduler process. This process is responsible for activating the Operator Processes used to execute the nodes of a compiled query tree. Scheduler processes can be run on any processor, insuring that no processor becomes a bottleneck. In practice, however, scheduler processes consume almost no resources and it is possible to run a large number of them on a single processor.
- For each operator in a query tree, at least one Operator Process is employed at each processor participating in the execution of the operator. These operators are primed at system initialization time in order to avoid the overhead of starting processes at query execution time

Schedulers are used for multi-node transactions and coordinate 2PC for the transactions. The operators actually execute the transactions.

For query execution, the optimization process is somewhat simplified as Gamma only employs hash-based algorithms for joins and other complex operations. Queries are compiled into a left-deep tree of operators. In the case of a single site query, the query is sent directly by the QM to the appropriate processor for execution. In the case of a multiple site query, the optimizer establishes a connection to an idle scheduler process through a

centralized dispatcher process. The dispatcher process, by controlling the number of active schedulers, implements a simple load control mechanism. Once it has established a connection with a scheduler process, the QM sends the compiled query to the scheduler process and waits for the query to complete execution. The scheduler process, in turn, activates operator processes at each query processor selected to execute the operator. Finally, the QM reads the results of the query and returns them through the ad-hoc query interface to the user or through the embedded query interface to the program from which the query was initiated.

The algorithms for all the relational operators are written as if they were to be run on a single processor. The input to an Operator Process is a stream of tuples and the output is a stream of tuples that is demultiplexed through a structure we term a split table. Once the process begins execution, it continuously reads tuples from its input stream, operates on each tuple, and uses a split table to route the resulting tuple to the process indicated in the split table. When the process detects the end of its input stream, it first closes the output streams and then sends a control message to its scheduler process indicating that it has completed execution. Closing the output streams has the side effect of sending ”end of stream” messages to each of the destination processes.

Gamma is built on top of an operating system designed specifically for supporting database management systems. NOSE provides multiple, lightweight processes with shared memory. A non-preemptive scheduling policy is used to help prevent convoys from occurring. . File services in NOSE are based on the Wisconsin Storage System (WiSS), where critical sections are protected via semaphores.

1.5.2 Query Processing

Gamma supports sort merge join, grace hash join, simple hash join, and hybrid hash join. Theta joins are not supported. [Shreya: what is a theta join?]

The multiprocessor join algorithms provided by Gamma are based on concept of partitioning the two relations to be joined into disjoint subsets called buckets by applying a hash function to the join attribute of each tuple. The partitioned buckets represent disjoint subsets of the original relations and have the important characteristic that all tuples with the same join attribute value are in the same bucket. **This study found that the Hybrid hash join almost always provides the best performance, it is now the default algorithm in Gamma.** In a hybrid hash join, the join broken up into many buckets, where tuples in one bucket fit in memory at a time. The size of the smaller relation determines the number of buckets.

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy	r7	r0	r1	r2	r3	r4	r5	r6

Chained Declustering (Relation Cluster Size = 8)

1.5.3 Transaction and Failure Management

Nodes use 2PL—two lock granularities, file, and page, and five lock modes, S, X, IS, IX, and SIX are provided. Each node in Gamma has its own local lock manager and deadlock detector. The lock manager maintains a lock table and a transaction wait-for-graph. The cost of setting a lock varies from approximately 100 instructions, if there is no conflict, to 250 instructions if the lock request conflicts with the granted group. In this case, the wait-for-graph must be checked for deadlock and the transaction that requested the lock must be suspended via a semaphore mechanism.

Nodes perform deadlock detection locally and periodically sent waits-for graphs to a centralized deadlock detector. Initially, the period for running the centralized deadlock detector is set at one second. Each time the deadlock detector fails to find a global deadlock, this interval is doubled and each time a deadlock is found the current value of the interval is halved. . The upper bound of the interval is limited to 60 seconds and the lower bound is 1 second. Whenever a cycle is detected in the global wait-for-graph, the centralized deadlock manager chooses to abort the transaction holding the fewest number of locks.

For recovery, nodes perform ARIES in a distributed fashion. Nodes don't write their logs locally—they send it to a remote node. This means there is a cost in sending messages to the remote node before persisting data. If M is the number of log processors being used, query processor i will direct its log records to the (i mod M) log processor. The scheduler process coordinates 2PC among the ARIES processes that are a part of a transaction.

Gamma uses chained declustering as opposed to interleaved declustering. See Figures 1.5.3 and 1.5.3 respectively to see the difference. With the interleaved declustering mechanism the set of disks are divided into units of size N called clusters. Each backup fragment is subdivided into N-1 subfragments and each subfragment is placed on a different disk within the same cluster other than the disk containing the primary

Chained and interleaved declustering can both sustain failure of a single disk or processor. Both strategies uniformly distribute the workload of the cluster. For interleaved clustering, one might conclude that the cluster size should be made as large as possible; until, of course, the overhead of the parallelism starts to overshadow the benefits obtained. While this is true for chained declustering, the availability of the interleaved strategy is inversely proportional to the cluster size. since the failure of any two processors or disk will render data unavailable. Thus, doubling the cluster size in order to halve

	cluster 0				cluster 1			
Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy		r0.0	r0.1	r0.2		r4.0	r4.1	r4.2
	r1.2		r1.0	r1.1	r5.2		r5.0	r5.1
	r2.1	r2.2		r2.0	r6.1	r6.2		r6.0
	r3.0	r3.1	r3.2		r7.0	r7.1	r7.2	

Interleaved Declustering (Cluster Size = 4)

(approximately) the increase in the load on the remaining nodes when a failure occurs has the (quite negative) side effect of doubling the probability that data will actually be unavailable.

During the normal mode of operation, read requests are directed to the fragments of the primary copy and write operations update both copies. When a failure occurs, pieces of both the primary and backup fragments are used for read operations. For example, with the failure of node 1, primary fragment R1 can no longer be accessed and thus its backup fragment r1 on node 2 must be used for processing queries that would normally have been directed to R1. However, instead of requiring node 2 to process all accesses to both R2 and r1, chained declustering offloads 6/7-ths of the accesses to R2 by redirecting them to r2 at node 3. In turn, 5/7- ths of access to R3 at node 3 are sent to R4 instead. This dynamic reassignment of the workload results in an increase of 1/7-th in the workload of each remaining node in the cluster. This is a contrived example though—in reality, queries cannot simply access an arbitrary fraction of a data fragment, especially given the variety of partitioning and index mechanisms provided by the Gamma software.

1.5.4 Lessons Learned

Lots of nice software engineering principles came into play. For example, a global index is a union of local indexes. Split tables could route outputs (single program; multiple data). Admission control (preventing DBMS from having more workload than it can take on) could happen at dispatch time. It separated log processes and query processes.

One big lesson learned was to balance the I/O workload, not necessarily the data volume. This meant dynamic data placement!

Their performance evaluation was great. They evaluated on 100k, 1M, and 10M tuples in a relation. They assessed both speedup (fix data size, vary number of nodes) and scaleup (vary data and node sizes proportionally). Speedup for selection and join queries were linear! For scaleup, they also got promising results—constant response time can be maintained for both selection and join queries as data and nodes were increased proportionally. Also, when they didn't get what they wanted, they explained why. For example,

typically at small scale they enjoy minimal disk seeks, but as they scale up they claim that disk seeks kick in. Non-linearity in magnetic disks is hard to avoid (”the source of the superlinear speedups exhibited by these queries is due to significant differences in the time the various configurations spend seeking”).

2 Query Processing

In this section, we discuss the following papers:

- Access Path Selection in a Relational Database Management System
- Query Evaluation Techniques for Large Databases
- The Volcano Optimizer Generator: Extensibility and Efficient Search
- Eddies: Continuously adaptive query processing
- Worst-Case Optimal Join Algorithms
- Datalog and Recursive Query Processing (Sections 1-3, 6)

2.1 Access Path Selection in a Relational Database Management System [47]

Abstract

In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates. System R is an experimental database management system developed to carry out research on the relational model of data. System R was designed and built by members of the IBM San Jose Research Laboratory.

This System R optimizer uses selectivity heuristics (i.e., how many tuples will be returned by a predicate) and appropriately weighted CPU and disk costs or accessing these tuples. It is a ”bottom-up” algorithm that uses dynamic programming and restricts to left-deep plans. The cost of a query plan is `PAGE FETCHES + w * RSI CALLS`, where the former term is the I/O and the latter term is an approximation of CPU time (i.e., number of tuples returned).

Question 2.1. What does System R consider in its optimizer algorthm and why?

2.1.1 Processing of a SQL Statement

The four phases of statement processing are parsing, optimization, code generation, and execution. The parser checks for correct syntax. The optimizer accumulates the names of tables and columns referenced in a query and looks them up in System R catalogs. It also checks for semantic errors and type compatibility based on metadata from the catalog, then performs access path selection. After a plan is chosen for each query block, the code generator translates it into machine language code. Then this code is executed by making calls via the RSI (storage system interface) to the RSS (internal storage system).

2.1.2 Research Storage System

The RSS maintains relations, access paths, locking, logging, and recovery features. It presents a tuple-oriented interface (RSI) to its users. Relations are stored in the RSS as a collection of tuples whose columns are physically contiguous. These tuples are stored on 4K byte pages; no tuple spans a page. Pages are organized into logical units called segments. Segments may contain one or more relations, but no relation may span a segment.

The primary way of accessing tuples in a relation is via an RSS scan. A scan returns a tuple at a time along a given access path. OPEN, NEXT, and CLOSE are the principal commands on a scan. There are 2 types of scans: segment scans find all the tuples of a given relation, and index scans to read the leaf pages of a B-tree and fetch the corresponding tuples. In a segment scan, all the non-empty pages of a segment are touched regardless of whether every tuple on the page is desired in the query. In the index scan, each page in the index is touched only once, but a data page might be touched multiple times if the index is unclustered.

Definition 2.1. A **clustered index** is one that maintains physical page proximity for tuples close together in the index pages. A clustered index has the property that not only each index page, but also each data page containing a tuple from that relation will be touched only once in a scan on that index.

Both index and segment scans may optionally take a set of predicates, called search arguments (or SARGS), which are applied to a tuple before it is returned to the RSI caller. A sargable predicate is one that can be expressed in the form ”column comparison-operator value,” e.g., age > 5.

2.1.3 Costs for Single Relation Access Paths

The optimizer uses the following cost prediction formula:

```
COST = PAGE FETCHES + W * (RSI CALLS)
```

RSI CALLS is the predicted number of tuples returned from the RSS. Since most of System R's CPU time is spent in the RSS, the number of RSI calls is a good approximation for CPU utilization. Thus the choice of a minimum cost path to process a query attempts to minimize total resources required. We say that a predicate or set of predicates matches an index access path when the predicates are sargable and the columns mentioned in the predicate(s) are an initial substring of the set of columns of the index key. For example, a NAME, LOCATION index matches NAME = 'SMITH' AND LOCATION = 'SAN JOSE'. If an index matches a boolean factor, an access using that index is an efficient way to satisfy the boolean factor. Sargable boolean factors can also be efficiently satisfied if they are expressed as search arguments. Note that a boolean factor may be an entire tree of predicates headed by an OR.

For looking up information about relations, the catalog contains the following information:

- `NCARD(T)`: cardinality of relation T
- `TCARD(T)`: number of pages in the segment that holds tuples of relation T
- `P(T)`: fraction of data pages in the segment that holds tuples of relation T. `P(T) = TCARD(T) /` number of non-empty pages in the segment
- `ICARD(I)`: number of distinct keys in index I
- `NINDX(I)`: number of pages in index I

Initial relation loading and index creation initialize these statistics. They are then updated periodically by an UPDATE STATISTICS command, which can be run by any user. System R does not update these statistics at every INSERT, DELETE, or UPDATE because of the extra database operations and the locking bottleneck this would create at the system catalogs.

Using the statistics, we can derive the selectivity factor for each boolean factor in the predicate list. See Figure 2. We assume that a lack of statistics implies that the relation is small, so an arbitrary factor is chosen. Another assumption (probably poor one) is that tuples are evenly distributed among index key values.

TABLE 1 SELECTIVITY FACTORS

```
column = value
         F = 1 / ICARD(column index) if there is an index on column
         This assumes an even distribution of tuples among the index key values.
         F = 1/10  otherwise

column1 = column2
         F = 1/MAX(ICARD(column1 index), ICARD(column2 index))
             if there are indexes on both column1 and column2
         This assumes that each key value in the index with the smaller cardinality has a
         matching value in the other index.
         F = 1/ICARD(column-i index) if there is only an index on column-i
         F = 1/10  otherwise

column > value (or any other open-ended comparison)
         F = (high key value - value) / (high key value - low key value)
         Linear interpolation of the value within the range of key values yields F if the col-
         umn is an arithmetic type and value is known at access path selection time.
         F = 1/3  otherwise (i.e. column not arithmetic)
         There is no significance to this number, other than the fact that it is less selec-
         tive than the guesses for equal predicates for which there are no indexes, and that
         it is less than 1/2. We hypothesize that few queries use predicates that are satis-
         fied by more than half the tuples.

column BETWEEN value1 AND value2
         F = (value2 - value1) / (high key value - low key value)
         A ratio of the BETWEEN value range to the entire key value range is used as the
         selectivity factor if column is arithmetic and both value1 and value2 are known at
         access path selection.
         F = 1/4  otherwise
         Again there is no significance to this choice except that it is between the default
         selectivity factors for an equal predicate and a range predicate.

column IN (list of values)
         F = (number of items in list) * (selectivity factor for column = value)
         This is allowed to be no more than 1/2.

columnA IN subquery
         F = (expected cardinality of the subquery result) / (product of the cardinalities of
              all the relations in the subquery's FROM-list).
         The computation of query cardinality will be discussed below. This formula is derived
         by the following argument:
         Consider the simplest case, where subquery is of the form "SELECT columnB FROM rela-
         tionC ...". Assume that the set of all columnB values in relationC contains the set
         of all columnA values. If all the tuples of relationC are selected by the subquery,
         then the predicate is always TRUE and F = 1. If the tuples of the subquery are
         restricted by a selectivity factor F', then assume that the set of unique values in
         the subquery result that match columnA values is proportionately restricted, i.e. the
         selectivity factor for the predicate should be F'. F' is the product of all the sub-
         query's selectivity factors, namely (subquery cardinality) / (cardinality of all pos-
         sible subquery answers). With a little optimism, we can extend this reasoning to
         include subqueries which are joins and subqueries in which columnB is replaced by an
         arithmetic expression involving column names. This leads to the formula given above.

(pred expression1) OR (pred expression2)
         F = F(pred1) + F(pred2) - F(pred1) * F(pred2)

(pred1) AND (pred2)
         F = F(pred1) * F(pred2)
         Note that this assumes that column values are independent.

NOT pred
         F = 1 - F(pred)
```


Figure 2: Selectivity Factors

Question 2.2. What are selectivity factors for operations in Figure 2?

Query cardinality (QCARD) is the product of the cardinalities of every relation in the query block's FROM list times the product of all the selectivity factors of that query block's boolean factors. The number of expected RSI calls (RSICARD) is the product of the relation cardinalities times the selectivity factors of the sargable boolean factors, since the sargable boolean factors will be put into search arguments which will filter out tuples without returning across the RSS interface.

We say that a tuple order is an interesting order if that order is one specified by the query block's GROUP BY or ORDER BY clauses. For single relations, the cheapest access path is obtained by evaluating the cost for each available access path (each index on the relation, plus a segment scan). . To find the cheapest access plan for a single relation query, we need only to examine the cheapest access path which produces tuples in each "interesting" order and the cheapest "unordered" access path.

2.1.4 Access Path Selection for Joins

For joins involving two relations, the two relations are called the outer relation, from which a tuple will be retrieved first, and the inner relation, from which tuples will be retrieved, possibly depending on the values obtained in the outer relation tuple.

There are 2 join methods: nested loops and merging scans. N-way joins can be visualized as a sequence of 2-way joins. At each step, we need to identify the best outer and inner relations. Although the cardinality of the join of n relations is the same regardless of join order, the cost of joining in different orders can be substantially different. There are n factorial permutations of join orders. Easy ways to reduce the search space are:

- Defer Cartesian products until the very end of the query plan
- Using join predicates on at least one relation with an interesting order
- Considering only "left-deep" plans

The cost of a nested loop or merge join is:

`OUTERCOST(path1) + N * INNERCOST(path2)`, where N is the product of the cardinalities and selectivity factors on all relations joined so far

It is interesting to observe that the cost formula for nested loop joins and the cost formula for merging scans are essentially the same. The reason that merging scans is sometimes better than nested loops is that the cost of the inner scan may be much less. There are 2^n subsets of the n relations, so the number of solutions that must be stored at most is $2^n \times$ number of interesting result orders.

2.1.5 Nested Queries

The optimizer evaluates the subquery before the top-level queries. A subquery may contain a reference to a value obtained from a candidate tuple of a higher level query block (see example below). Such a query is called a correlation subquery. In this case, for each candidate tuple of the top level query block, the value is used for evaluation of the subquery. In other words, correlated subqueries are evaluated once per outer tuple.

2.2 Query Evaluation Techniques for Large Databases [24]

Abstract

Database management systems will continue to manage large data volumes. Thus, efficient algorithms for accessing and manipulating large sets and sequences will be required to provide acceptable performance. The advent of object-oriented and extensible database systems will not solve this problem. On the contrary, modern data models exacerbate the problem: In order to manipulate large sets of complex objects as efficiently as today's database systems manipulate simple records, query processing algorithms and software will become more complex, and a solid understanding of algorithm and architectural issues is essential for the designer of database management software. This survey provides a foundation for the design and implementation of query execution facilities in new database management systems. It describes a wide array of practical query evaluation techniques for both relational and postrelational database systems, including iterative execution of complex query evaluation plans, the duality of sort- and hash-based set-matching algorithms, types of parallel query execution and their implementation, and special operators for emerging database application domains.

This is a beast of a paper with a survey of query evaluation techniques based on sorting, hashing, and partitioning. Plans are assumed to be optimized; this paper focuses on executing the machine code. The paper talks about "large data sets" as a motivation for some of the techniques, but exact or specific semantics of the large data sets may dictate how techniques will perform in reality - e.g. multimedia video data v.s. human genome data v.s. multi-dimensional geographic data probably all have individual optimizations that affect these techniques, but such possible optimizations are not discussed.

Also check out http://aturing.umcs.maine.edu/~sudarshan.chawathe/200509/cos480/notes/qeval.pdf for some review questions.

2.2.1 Architecture of Query Execution Engines

The query execution engine is a collection of query execution operators and mechanisms for operator communication and synchronization.

Different systems may implement the same data model and the same logical algebra but may use very different physical algebras. For example, while one relational system may use only nested-loops joins, another system may provide both nested-loops join and merge-join, while a third one may rely entirely on hash join algorithms.

Another significant difference between logical and physical algebras is the fact that specific algorithms and therefore cost functions are associated only with physical operators, not with logical algebra operators. On the physical or representation level, there is typically a smaller set of representation types and structures, e.g., file, record, record identifier (RID), and maybe very large byte arrays. Query optimization is the mapping from logical to physical operations, and the query execution engine is the implementation of operations on physical representation types and of mechanisms for coordination and cooperation among multiple such operations in complex queries.

Synchronization and data transfer between operators is the main issue to be addressed in the architecture of the query execution engine. For example, imagine a query with 2 joins. How do we pass the result of the first join to the second one? The simplest method is to create a temporary file. Another option is to use inter process communication (e.g., pipes) to transfer data between operators. One paper proposes avoiding both altogether and rewriting a tree of operators into a single iterative program with nested loops and other control structures. The most practical alternative is to implement all operators in such a way that they schedule each other within a single operating system process. We define a **granule**, or a single record typically, and iterate over all granules comprising an intermediate query result. [Shreya: I don't understand this?] Each time an operator needs another piece of data ("granule"), it calls its data input operator's next function to produce one. Operators structured in such a manner are called iterators.

The interesting observations are that (i) the entire query plan is executed within a single process, (ii) operators produce one item at a time on request, (iii) this model effectively implements, within a single process, (special-purpose) coroutines and demand-driven data flow. (iv) items never wait in a temporary file or buffer between operators because they are never m-educed before they are needed, (v) therefore this model is very efficient in its time-space-product memory costs, (vi) iterators can schedule any tree. including bushy trees (see below), (vii) no operator is affected by the complexity of the whole plan, i.e., this model of operator implementation and synchronization works for simple as well as very complex query plan.

Query plans are algebra expressions and can be represented as trees. Left-deep (every right subtree is a leaf), right-deep (every left-subtree is a leaf), and bushy (arbitrary) are the three common structures. In a left-deep tree, each operator draws input from one input, and an inner loop interates over the other input. Iterators are relatively straightforward to implement and are suitable building blocks for efficient, extensible query processing engines.

2.2.2 Sorting

Most DBMSes use merging because large data sets are expected. Only in the unusual case that a data set is smaller than the available memory can in memory techniques such as quicksort be used. Sorting modules' interfaces follow the structure of iterators.

All sort algorithms try to exploit the duality between main-memory mergesort and quicksort. Both of these algorithms are recursive divide-and-conquer algorithms. The difference is that merge sort first divides physically and then merges on logical keys, whereas quicksort first divides on logical keys and then combines physically by trivially concatenating sorted subarrays. A point of practical importance is the fan-in or degree of merging, but this is a parameter rather than a defining algorithm property.

Quicksort and replacement-selection are the two algorithms of choice for creating the set of initial (level-0) runs. Replacement selection fills memory in a priority heap, smallest key is written to a run and replaced from next in input; this replacement will probably be bigger than the just written item, so we can then iterate. If not, put mark replacement for next run file. Quicksort has bursty I/O pattern, RS smoothly alternates between read and write operations. If only a single device is used, quicksort may result in faster I/0 because fewer disk arm movements are required. Replacement selection usually has fewer and longer run files.

The problem with replacement selection is memory management. If input items are kept in their original pages in the buffer (in order to save copying data, a real concern for large data volumes) each page must be kept in the buffer until its last record has been written to a run file. On average, half a page's records will be in the priority heap. The solution to this problem is to copy records into a holding space and to keep them there while they are in the priority heap and until they are written to a run file.

The level-O runs are merged into level1 runs, which are merged into level-2 runs, etc., to produce the sorted output. The maximal merge fan-in $F = \lfloor M/C - 1 \rfloor$ is the number of runs that can be merged in one time, where M is memory size (pages) and C is cluster or unit of I/O (pages). Since the sizes of runs grow by a factor of F from level to level, the number of merge levels L is $\log_F(W)$, where W is the number of level-0 run files.

Question 2.3. How are sorted runs merged? How many merge levels do you need to have to merge an entire run?

Here are some tricks to improve efficiency:

- Scans are faster if read-ahead and write-behind are used; therefore, double-buffering using two pages of memory per input run and two for the merge output might speed the merge process. We also do not need to double buffer every input run. We can look at the high key of every page from every run to see which one will get buffered out first.

Definition 2.2. Double buffering is when data in one buffer is being processed, while the next set of data is read into the other buffer. It helps overlap I/O with processing.

- Using large cluster sizes for the run files is very beneficial. Larger cluster sizes will reduce the fan-in and therefore may increase the number of merge levels, but this is ok: merging levels are performed faster because fewer I/O operations and disk seeks and latency delays are required. Determine optimal cluster size either by physical disk characteristics, or by matching processor and I/O latencies.
- Some operators require multiple sorted inputs, so memory must be divided among multiple final merges; this final fan-in and the normal fan-in of each sort should be specified separately.
- If we're sorting a relation that's only slightly bigger than memory, we can write out a small sorted run to disk, then sort everything else in memory, and then merge the two together.

Note 2.1. Total I/O cost vs cluster size usually forms a u-shape, meaning the optimal cluster size is somewhere in the middle.

2.2.3 Hashing

[Shreya: What is fan out here?]

Hashing-based query processing algos use in-memory hash table of database objects; if data in hash table is bigger than main memory (common case), then hash table overflow occurs. Three techniques for overflow handling exist:

- Avoidance: input set is partitioned into F files before any in-memory hash table is built. Partitions can be dealt with independently. Partition sizes must be chosen well, or recursive partitioning will be needed.
- Resolution: assume overflow won't occur; if it does, partition dynamically.
- Hybrid: like resolution, but when partition, only write one partition to disk, keep the rest in memory.

Quality of hash function is key, so you don't end up with skewed recursive partition depths, leading to poor performance and much disk I/O. There are three ways to assign hash buckets to partitions:

- Each time an overflow occurs, just assign a fixed number of buckets to a new partition. In Gamma, the number of disk partitions is chosen such that each bucket can reasonably be expected to fit in memory.
- Dynamic destaging: a large number of small partition files in created and then collapsed into fewer partition files no larger than memory. Set the cluster size very small and fan out large. When an overflow occurs, assign the largest partition to disk.
- Gather statistics before partitioning.

2.2.4 Disk Access

[Shreya: Go over B, B+ etc trees]

File scans can be made fast with read-ahead (track-at-a-crack). Requires contiguous file allocation, so may need to bypass OS/file system.

Leafs of indices should point to data records, not contain them, so one can separate record lookup from index scanning. Advantages:

- You can scan records without retrieving records
- Multiple indices can be joined
- You can take union or intersection of index scans

Cache data in I/O buffer. LRU is wrong for many database operations. For example, a file scan reads a large set of pages but uses them only once, "sweeping" the buffer clean of all other pages, even if they might be useful in the future and should be kept in memory. Buffer management mechanisms usually provide fix/unfix (pin/unpin) semantics on a buffer page, which iterator implementations can take advantage of when passing buffer pages amongst themselves.

Question 2.4. Why is LRU wrong for many database operations?

2.2.5 Aggregation and Duplicate Removal

Aggregate functions require grouping; this grouping is virtually identical to what is required in duplicate removal, so aggregation and duplicate removal are treated as the same. There are multiple ways to do this:

- Nested loop aggregation: naive method: using temporary file to accumulate output, loop for each input item over output file, and either aggregate in the new item into appropriate output item, or append new output item to file. Inefficient for large data sets.
- Aggregation based on sorting: sorting brings items together. Use to detect and remove duplicates or do aggregation as soon as possible; implement in routines that create the run files. "Early aggregation".
- Aggregation based on hashing: if hash function puts items of same group in same bucket (or nearby bucket), duplicate removal/aggregatation can be done when inserting into hash table.

Early aggregation is a big win, performance-wise, as size of run files or hash table is reduced. Both sort and hash result in logarithmic performance based on input data set size. For both sort based and hash based aggregation and duplicate removal, we can start performing deduplication and aggregation as the algorithms run (to reduce run sizes in future passes); we do not have to wait for the final sort or grouping.

Question 2.5. In sorting and hashing aggregation, why would we want to start performing aggregation as early as possible?

2.2.6 Binary Matching Operators

Aggregation condenses information. But how do we combine information (i.e., join)? Most systems today use nested-loop join or merge join, since System R found that these always gave the best or close to the best performance. Hash joins have recently been studied with some interest, though.

Nested loop join. Scan "inner" input once for each value of "outer" input, looking for a match. This gives horrible performance. There are some basic optimizations. For one-to-one match, can terminate inner scan as soon as match found. You can also do block nested loop join (scan inner pages for each block of outer pages). You could also save inner pages in memory. Alternatively, you could scan inner input backwards and forwards, but this only saves the last page or two.

Index nested loop join. Exploit a permanent or temporary index on the inner input's join attribute to replace file scans by index lookups. This can be way fast under certain conditions (index depth * size of smaller input $<$ size of larger input). Another interesting idea using two ordered indices, e.g., a B-tree on each of the two join columns, is to switch roles of inner and outer join inputs after each index lookup, which leads to the name "zig-zag join." It is not immediately clear under which circumstances this join method is most efficient.

Merge join. This requires both inputs to be sorted on join attribute, then scanned by stepping a pair of pointers. It is efficient if the inputs are produced by a previous merge-join step, since that means they're already sorted. A combination of nested-loops join and merge-join is the heap-filter merge-join. It first sorts the smaller inner input by the join attribute and saves it in a temporary file. Next, it uses all available memory to create sorted runs from the larger outer input using replacement selection (there will be about $W = R/(2 \times M) + 1$ runs). These runs are not written to disk; instead, they are joined immediately with the sorted inner input using merge-join. Thus, the number of scans of the inner input is reduced to about one half when compared to block nested loops. But block nested loops don't read and write temporary files for the outer inputs.

Hash join. Build in-memory hash table on one input, probe it with items from the other. However, they require overflow avoidance or resolution methods for larger build inputs. If inputs too large, they're recursively partitioned. Outputs from each partitioning are concatenated. Cost is roughly logarithmic in input size; main difference compared to merge join is that the recursion depth depends only on the build input, not both inputs.

Pointer based join. Pointer based joins typically outperform value joins if only a small fraction of the outer input actually participates in the join. When S-records point to R-records, the cost of the pointer join is even higher than for nested-loops join. The variants are flavors of other join algorithms (nested loop, merge-join, and hybrid hash join). This honestly feels like another index, though.

Question 2.6. Discuss 3 join operator algorithms and pros and cons of each.

Question 2.7. What's the difference between hashing and hybrid hashing algorithms?

2.2.7 Universal Quantification

Universal quantification permits queries such as "find the students who have taken all database courses." In the past, universal quantification has been largely ignored for four reasons. First, typical database applications, e.g., record-keeping and accounting applications, rarely require universal quantification. Second, it can be circumvented using a complex expression involving a Cartesian product. Third, it can be circumvented using complex aggregation expressions. Fourth, there seemed to be a lack of efficient algorithms.

We can formulate this problem as: R(A, B) / S(B) should return the set of R.A which is paired with every entry in S.B. Typically, universal quantification can easily be replaced by aggregations. (Intuitively, all universal quantification can be replaced by aggregation. However, we have not found a proof for this statement.) For example, the example query about database courses can be restated as "find the students who have taken as many database courses as there are database courses."

There are four methods to compute the quotient of two relations, a sort- and a hash-based direct method, and sort- and hash-based aggregation.

Sort-based division. This sorts the divisor input on all its attributes and the dividend relation with the quotient attributes as major and the divisor attributes as minor sort keys. Then we scan values of the dividend to make sure the divisor fully exists for the dividend keys.

Hash-based division. uses two hash tables, one for the divisor and one for the quotient candidates. While building the divisor table, a unique sequence number is assigned to each divisor item. After the divisor table has been built, the dividend is consumed. [Shreya: What is this? How to handle duplicates?]

Question 2.8. Discuss 2 reasons why supporting universal quantification operators might have been ignored in the past.

2.2.8 Duality of Sort and Hash

Both sorting and hashing divide and recombine things. If a data set fits into memory, quicksort is the sort-based method to manage data sets while classic (in-memory) hashing can be used as a hashing technique. It is interesting to note that both quicksort and classic hashing are also used in memory to operate on subsets after "cutting" an entire large data set into pieces. But there are differences. In the sort-based algorithms, a large data set is divided into subsets using a physical rule, namely into chunks as large as memory. These chunks are later combined using a logical step, merging. In the hash-based algorithms, large inputs are cut into subsets using a logical rule, by hash values. The resulting partitions are later combined using a physical step, i.e., by simply concatenating the subsets or result subsets.

If merging is done in the most naive way, i.e., merging all runs of a level as soon as their number reaches the fan-in, the last merge on each level might not be optimal. Similarly, if the highest possible fan-out is used in each partitioning step, the partition files in the deepest recursion level might be smaller than memory, and less than the entire memory is used when processing these files. Thus, in both approaches the memory resources are not used optimally in the most naive versions of the algorithms.

The development of hybrid hash algorithms was a consequence of the advent of large main memories that had led to the consideration of hash-based join algorithms in the first place. Another well-known technique to use memory more effectively and to improve sort performance is to generate runs twice as large as main memory using a priority heap for replacement selection. [Shreya: how does this work?]

In many sort implementations, namely those using eager merging, the first and second phase interleave as a merge step is initiated whenever the number of runs on one level becomes equal to the fan-in. Similarly, a grace hash join can recursively partition depth-first or breadth-first. Depth-first partitioning reduces the time to first output.

Hash-based algorithms tend to produce their outputs in a very unpredictable order, depending on the hash function and on overflow management. In order to take advantage of multiple joins on the same attribute (or of multiple intersections, etc.) similar to the advantage derived from interesting orderings in sort-based query processing, the equality of attributes has to be exploited during the logical step of hashing, i.e., during partitioning.

Question 2.9. If data fits in memory, what's the best sorting technique? Hashing technique?

Question 2.10. Name a similarity and difference between sorting and hashing.

2.2.9 Execution of Complex Query Plans

In early relational execution engines, optimal scheduling of multiple operators and the division and allocation of resources in a complex plan was largely ignored for 2 reasons: (1) only left-deep trees were considered—i.e., the right had to be a scan, making it impossible to do concurrent execution of multiple subplans in a single query, and (2) datasets were less parallelizable—e.g., sorting for nontrivial file sizes required the entire input to be written to temporary storage.

In other words, left-deep query plans limit the amount of resource contention between operators. If all the operators pipeline, then likely each one doesn't have that much state. If one of the operators blocks, then its parents don't contend with it.

For right-deep trees, you can build hash tables concurrently. Through effective use of bit filtering, memory requirements of right-deep plans might be comparable to those of left deep plans. One paper interprets and executes bushy plans as multiple right-deep subplans.

When multiple operators compete for resources, it is important to allocate the number of buffer pages to each operator proportional to the amount of data they have to process.

Optimizing big trees can be ineffective because it can be hard get accurate selectivity estimates. Ingres did a trick called decomposition in which subtrees of the big plan were optimized and executed incrementally. The justification and advantage of this approach are that all earlier selectivities are known for each decision, because the intermediate results are materialized. The disadvantage is that data flow between operators cannot be exploited, resulting in a significant cost for writing and reading intermediate files. Ingres did only one selection or join at a time, but you can modify this to execute multiple steps in each cycle at a time.

2.2.10 Mechanisms for Parallel Query Execution

The goal of parallel algorithms and systems is to obtain speedup and scaleup. Speedup considers additional hardware resources for a constant problem size; linear speedup is considered optimal. An alternative measure for a parallel system's design and implementation is scaleup, in which the problem size is altered with the resources. Similarly, linear scaleup is optimal.

A third measure of sucess for a parallel algorithm is based on Amdahl's law: the fraction of the sequential program for which linear speedup was attained, defined by $p = f \times s/d + (1 - f) \times s$ for sequential execution time s, parallel execution time p, and degree of parallelism d.

Question 2.11. Define speedup and scaleup.

In the database literature, "distributed" usually implies "locally autonomous," i.e., each participating system is a complete database management system in itself, with access control, metadata (catalogs), query processing, etc. In parallel systems, on the other hand, there is only one locus of control. In other words, there is only one database management system that divides individual queries into fragments and executes the fragments in parallel. The three distributed architectures are shared memory, shared disk, and shared nothing (and shared nothing with each node being shared memory). Stonebraker thinks shared-nothing architecture in distributed systems is best; the authors think hierarchical computer architectures are best (consisting of multiple clusters, each with multiple CPUS and disks and a large shared memory).

There are multiple forms of parallelism. Interquery parallelism is a direct result of the fact that most database management systems can service multiple requests concurrently. Interoperator parallelism is basically pipelining, or parallel execution of different operators in a single query. Interoperator parallelism can be used in two forms, either to execute producers and consumers in pipelines, called vertical interoperator parallelism here, or to execute independent subtrees in a complex bushy-query evaluation plan concurrently, called horizontal interoperator or bushy parallelism here (e.g., merge join receiving input from two sort processes). The main problem with bushy parallelism is that it is hard or impossible to ensure that the two subplans start generating data at the right time and generate them at the right rates.

Question 2.12. Discuss the 4 forms of parallelism. Explain a challenge in supporting vertical interoperator parallelism (i.e., being able to execute operators from independent subtrees in the query plan).

The final form of parallelism in database query processing is intraoperator parallelism in which a single operator in a query plan is executed in multiple processes, typically on disjoint pieces of the data partitioned on different processors. But if the underlying data is sequential or time series, it would be hard to independently partition this data.

Question 2.13. When would intraoperator parallelism fail to work efficiently?

here are two general approaches to parallelizing a query execution engine, which we call the bracket and operator models and which are used, for example, in the Gamma and

Volcano systems, respectively. In the bracket model, there is a generic process template that can receive and send data and can execute exactly one operator at any point of time. So, every operator lives in its own process and communicates over the network. This is expensive, particularly when whole queries can be executed in a single process!

An alternative to the bracket model is the operator model. We insert "parallelism" operators into a sequential plan, called exchange operators. The exchange operator is an iterator like all other operators in the system with open, next, and close procedures; therefore, the other operators are entirely unaffected by the presence of exchange operators in a query evaluation plan. Exchange operators are essentially no-ops at the data level, but contributes to control flow.

Separation of data manipulation from process control and interprocess communication can be considered an important advantage of the operator model of parallel query processing, because it permits design, implementation, and execution of new data manipulation algorithms such as N-ary hybrid hash join without regard to the execution environment.

Question 2.14. What are advantages of the operator model over the bracket model of parallelizing a query execution engine?

The mapping of a sequential plan to a parallel plan by inserting exchange operators permits one process per operator as well as multiple processes for one operator (using data partitioning) or multiple operators per process, which is useful for executing a complex query plan with a moderate number of processes.

Skew management methods can be divided into basically two groups. First, skew avoidance methods rely on determining suitable partitioning rules before data is exchanged between processing nodes or processes. You can use quantiles for range partitioning. Second, skew resolution repartitions some or all of the data if an initial partitioning has resulted in skewed loads. Repartitioning is relatively easy in shared-memory machines, but can also be done in distributed-memory architectures, albeit at the expense of more network activity.

I/O, not processing, is the most likely bottleneck in high performance query execution.

2.2.11 Parallel Algorithms

Many algorithms need parallel versions of implementation. Most parallel sort algorithms consist of a local sort and a data exchange step. If the data exchange step is done first, quantiles must be known to ensure load balancing during the local sort step. We can

range partition initially and have remote nodes create sorted runs. Or we can sort locally and range partition sorted runs, so remote nodes merge instead of sort. We can build stats on the first pass to try and get better partitioning. If there is data skew, this approach can limit parallelism in the second pass. It can also lead to deadlock if you implement it really naively. Deadlock will occur if (1) multiple consumers feed multiple producers, (2) each producer produces a sorted stream, and each merges multiple sorted streams, (3) some key-based partitioning rule is used other than range partitioning, i.e., hash partitioning, (4) flow control is enabled, and (5) the data distribution is particularly unfortunate.

Question 2.15. Consider a parallel sorting algorithm where all the data is sorted locally, then sent to remote nodes for merging. Describe two ways in which deadlock can occur.

2.2.12 Nonstandard Query Processing Algorithms

[Shreya: TODO: later]

2.3 The Volcano Optimizer Generator: Extensibility and Efficient Search [26]

Abstract

Emerging database application domains demand not only new functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific database systems. One of these tools is a new optimizer generator. Data model, logical algebra, physical algebra, and optimization rules are translated by the optimizer generator into optimizer source code. Compared with our earlier EXODUS optimizer generator prototype, the search engine is more extensible and powerful; it provides effective support for non-trivial cost models and for physical properties such as sort order. At the same time, it is much more efficient as it combines dynamic programming, which until now had been used only for relational select-project-join optimization, with goal-directed search and branch-and-bound pruning. Compared with other rule-based optimization systems, it provides complete data model independence and more natural extensibility.

Volcano is an optimizer generator, which takes in information about the logical and physical algebra, translation rules, etc and outputs a query optimizer. It uses a top down dynamic programming query optimization algorithm. The requirements of Volcano were as follows:

- Usable with existing query execution software in other projects
- More efficient, both in runtime and memory consumption
- Provide extensible support for sort order and compression status and other physical properties
- Allow user-defined heuristics and data model semantics to guide the search and prune the search space
- Support flexible cost models that permit plans for incompletely specified queries

There are five fundamental design decisions embodied in the system, which contribute to the extensibility and search efficiency of optimizers designed and implemented with the Volcano optimizer generator.

1. Volcano optimizer generator uses two algebras, called the logical and the physical algebras, and generates optimizers that map an expression of the logical algebra (a query) into an expression of the physical algebra (a query evaluation plan consisting of algorithms)
2. Rules are translated independently from one another and are combined only by the search engine when optimizing a
3. No intermediate levels of algebra (probably to minimize cognitive overhead)
4. Compiled, not interpreted (because query optimization is CPU intensive)
5. Search engine based on dynamic programming

2.3.1 Optimizer Generator Input and Optimizer Operation

There are some operators in the physical algebra that do not correspond to any operator in the logical algebra, for example sorting and decompression. These are called enforcers.

Each optimization goal (and subgoal) is a pair of a logical expression and a physical property vector. In order to decide whether or not an algorithm or enforcer can be used to execute the root node of a logical expression, a generated optimizer matches the implementation rule, executes the condition code associated with the rule, and then invokes an applicability function that determines whether or not the algorithm or enforcer can deliver the logical expression with physical properties that satisfy the physical property vector.

The following are inputs to the optimizer generator:

- Logical operators (join, select, project, etc)
- Physical algorithms (tuple nested loop join, full table scan, etc)
- Logical to logical transformation rules (join is commutative, etc)
- Logical to physical transformation rules (join to sort merge join, etc)
- Enforcers: algorithms for sort, decompress, etc (no-ops, logically)
- Cost function for algorithms and enforcers, where cost is an abstract data type (estimated time, number of records, whatever unit the implementor wants)
- A property function for each logical and physical algebra expression. Logical properties encapsulate selectivity estimation, schemas, and more. Physical properties encapsulate sort order, etc.
- Applicability functions for each algorthm and enforcer (whether a given algorithm can implement an expression with the given physical properties and the physical requirements on its inputs)

2.3.2 The Search Engine

"Since our experience with the EXODUS optimizer generator indicated that it is easy to waste a lot of search effort in extensible query optimization, we designed the search algorithm for the Volcano optimizer generator to use dynamic programming and to be very goal-oriented, i.e., driven by needs rather than by possibilities."

The search strategy designed with the Volcano optimizer generator extends dynamic programming from relational join optimization to general algebraic query and request optimization and combines it with a top-down, goal-oriented control strategy for algebras in which the number of possible plans exceeds practical limits of precomputation. In a way, while the search engines of the EXODUS optimizer generator as well as of the System R and Starburst relational systems use forward chaining (in the sense in which this term is used in AI), the Volcano search algorithm uses backward chaining, because it explores only those subqueries and plans that truly participate in a larger expression.

Given a logical plan and desired set of physical properties, it creates a set of possible "moves" that satisfy the following:

1. Logical transformations (e.g., swapping join order)
2. Algorithms that give the required physical properties / logical to physical transformations (e.g., join to sort merge join)
3. Enforcers for the physical properties

It orders the set of moves by its promise or potential and recurses, memoizing the seen cost in a table that maps (logical expression, physical properties) to cheapest plan. Pursuing all moves or only a selected few isamajor heuristic placed into the hands of the optimizer implementor. In the extreme case, an optimizer implementor can choose to transform a logical expression without any algorithm selection and cost analysis, which covers the optimizations that in Starburst are separated into the query rewrite level. The difference between Starburst's two-level and Volcano's approach is that this separation is mandatory in Starburst while Volcano will leave it as achoice to be made by the optimizer implementor.

If a move to be pursued is a transformation, the new expression is formed and optimized using FindBestPlan. In order to detect the case that two (or more) rules are inverses of each other, the current expression and physical property vector is marked as "in progress." If a newly formed expression already exists in the hash table and is marked as "in progress," it is ignored because its optimal plan will be considered when it is finished. This is a trick to speed up search.

There are some other speed up tricks. For example, if a child of an operator costs more than the limit, we can stop looking at other children. Furthermore, cost limits are passed down in the optimization of subexpressions, and tight upper bounds also speed their optimization. It is important (for optimization speed, not for correctness) that a relatively good plan be found fast, even if the optimizer uses exhaustive search.

In summary, the search algorithm employed by optimizers created with the Volcano optimizer generator uses dynamic programming by storing all optimal subplans as well as optimization failures until a query is completely optimized.

Question 2.16. Compare and contrast the Volcano style optimizer with the Selinger-style System R optimizer. What are pros and cons of each?

2.3.3 Comparison with the EXODUS Optimizer Generator

Arguably, the most important contributions of the EXODUS optimizer generator are extensive use of algebraic rules (transformation rules and implementation rules), and its focus on software modularization. However, the EXODUS optimizer generator's search engine was far from optimal. First, the modifications required for unforeseen algebras and their peculiarities made it a bad patchwork of code. Second, the organization of the "MESH" data structure (which held all logical and physical algebra expressions explored so far) was extremely cumbersome, both in its time and space complexities. Third, the almost random transformations of expressions in MESH resulted in significant overhead in "reanalyzing" existing plans. In fact, for larger queries, most of the time was spent

reanalyzing existing plans.

Volcano makes a distinction between logical expressions and physical expressions. In EXODUS, only one type of node existed in the hash table called MESH, which contained both a logical operator such as join and a physical algorithm such as hybrid hash join. To retain equivalent plans using merge-join and hybrid hash join, the logical expression (or at least one node) had to be kept twice, resulting in a large number of nodes in MESH.

Physical properties were handled rather haphazardly in EXODUS. The ability to specify required physical properties and let these properties, together with the logical expression, drive the optimization process was entirely absent in EXODUS. In Volcano, we can include the cost of an enforcer as a step.

The concept of physical property is very powerful and extensible. The most obvious and well-known candidate for a physical property in database query processing is the sort order of intermediate results. Other properties can be defined by the optimizer implementor at will. Depending on the semantics of the data model, uniqueness might be a physical property with two enforcers, sort- and hash-based.

The Volcano algorithm is driven top-down; subexpressions are optimized only if warranted. In the extreme case, if the only move pursued is a transformation, a logical expression is transformed on the logical algebra level without optimizing its subexpressions and without performing algorithm selection and cost analysis for the subexpressions.

Cost is defined in much more general terms in Volcano than in the EXODUS optimizer generator.

Finally, the Volcano optimizer generator is more extensible than the EXODUS prototype, in particular with respect to the search strategy.

For the EXODUS-generated optimizer, the search effort increases dramatically from 3 to 4 input relations because reanalyzing becomes a substantial part of the query optimization effort in EXODUS at this point. The increase of Volcano's optimization costs is about exponential, shown in an almost straight line, which mirrors exactly the increase in the number of equivalent logical algebra expressions. TLDR: Volcano scales to multiple relations better.

In summary, Volcano is successful for multiple reasons:

- It bakes in the cost of enforcers into the search algorithm rather than tacking them onto the end, after optimizing the expression (in the Starburst approach)
- It is user-friendly, with no intermediate levels of algebra
- Top-down approach only considers optimizing subexpressions if needed (avoids unnecessary reanalysis like EXODUS)

- Separation of logical and physical algebras makes it easy to modify at either level (highly extensible)

Question 2.17. Discuss 3 things that contribute to Volcano's success.

2.4 Eddies: Continuously adaptive query processing [6]

In large federated and shared-nothing databases, resources can exhibit widely fluctuating characteristics. Assumptions made at the time a query is submitted will rarely hold throughout the duration of query processing. As a result, traditional static query optimization and execution techniques are ineffective in these environments.

In this paper we introduce a query processing mechanism called an eddy, which continuously reorders operators in a query plan as it runs. We characterize the moments of symmetry during which pipelined joins can be easily reordered, and the synchronization barriers that require inputs from different sources to be coordinated. By combining eddies with appropriate join algorithms, we merge the optimization and execution phases of query processing, allowing each tuple to have a flexible ordering of the query operators. This flexibility is controlled by a combination of fluid dynamics and a simple learning algorithm. Our initial implementation demonstrates promising results, with eddies performing nearly as well as a static optimizer/executor in static scenarios, and providing dramatic improvements in dynamic execution environments.

Query engines at scale need to function robustly in an unpredictable and constantly fluctuating environment, due to many reasons: hardware/workload complexity, data complexity, UI complexity, and more. Often the query plans chosen up front are not optimal when it comes time to execute the query, because selectivities are constantly changing. Eddy continuously reorders the application of pipelined operators in a query plan, on a tuple-by-tuple basis.

Because the eddy observes tuples entering and exiting the pipelined operators, it can adaptively change its routing to effect different operator orderings. A synchronization barrier occurs when one relation is stuck waiting for another (e.g. in a sort merge join, the relation with larger values waits for the relation with smaller values). A moment of symmetry occurs when the inner and outer relations of a join can be swapped (provided some extra bookkeeping is done). For example, (R join S) can be swapped after every complete iteration of S, so long as R advances its cursor and doesn't return tuples before it.

Intuitively we should have fewer synchronization barriers and more moments of symmetry for eddies to thrive. If we have some join R JOIN S in an eddy, we want to be able to shove in tuples from R and tuples from S in whatever order we desire. We don't want to be constrained to feed in one tuple of R and then all the tuples from S.

Question 2.18. In the context of eddies, what is a synchronization barrier? What is a moment of symmetry?

To change a query plan on the fly, a great deal of state in the various operators has to be considered, and arbitrary changes can require significant processing and code complexity to guarantee correct results. In a highly variable environment, the best-case scenario rarely exists for a significant length of time. So they favor adaptivity over best-case performance.

When a join algorithm reaches a barrier, it has declared the end of a scheduling dependency between its two input relations. In such cases, the order of the inputs to the join can often be changed without modifying any state in the join; when this is true, we refer to the barrier as a moment of symmetry. Moments of symmetry allow reordering of the inputs to a single binary operator. But we can generalize this, by noting that since joins commute, a tree of binary joins can be viewed as a single n-ary join. One could easily implement a doubly-nested-loops join operator over 3 relations, and it would have moments of complete symmetry at the end of each loop. At that point, all three inputs could be reordered with a straightforward extension to the discussion above: a cursor would be recorded for each input, and each loop would go from the recorded cursor position to the end of the input.

2.4.1 Ripple Joins

Eddies use ripple joins, a family of join algorithms invented for the online setting. Suppose you have 2 relations R and S. In a ripple join, you sample β_R elements from R and β_S elements from S at every time step. You join them with each other and with historical joined values. It is often necessary to sample one relation (the more variable" one) at a higher rate than another in order to provide the shortest possible confidence intervals for a given animation speed.

Ripple joins have moments of symmetry at each "corner" of a rectangular ripple, i.e., whenever a prefix of the input stream R has been joined with all tuples in a prefix of input stream S and vice versa. try. Ripple joins are attractive with respect to barriers as well. Ripple joins were designed to allow changing rates for each input; this was originally used to proactively expend more processing on the input relation with more statistical influence on intermediate results.

2.4.2 Rivers and Eddies

River is a dataflow query engine, analogous in many ways to Gamma. "Iterator"-style modules (query operators) communicate via a fixed dataflow graph (a query plan). Eddies are implemented in River. Although we will use eddies to reorder tables among joins, a heuristic pre-optimizer must choose how to initially pair off relations into joins, with the constraint that each relation participates in only one join.

An eddy is implemented via a module in a river containing an arbitrary number of input relations, a number of participating unary and binary modules, and a single output relation. An eddy encapsulates the scheduling of its participating operators; tuples entering the eddy can flow through its operators in a variety of orders. An eddy explicitly merges multiple unary and binary operators into a single n-ary operator within a query plan. An eddy contains a priority queue of tuples, where each tuple has an associated vector of "ready" or "done" bits (of size num operators). The eddy only ships out tuples to operators that have the "ready" bit turned on. After the operators have finished processing the tuple and returned it to the eddy, the eddy sets the "done" bit for the tuple. When all the "done" bits are set, the tuple is shipped to the eddy's output. When an eddy receives a tuple from one of its inputs, it zeroes the Done bits, and sets the Ready bits appropriately. When there are ordering constraints on the operators, the eddy turns on only the Ready bits corresponding to operators that can be executed initially.

2.4.3 Routing Tuples in Eddies

The routing policy used in the eddy determines the efficiency of the system. An eddy's tuple buffer is implemented as a priority queue with a flexible prioritization scheme. An operator is always given the highest-priority tuple in the buffer that has the corresponding Ready bit set. For simplicity, we start by considering a very simple priority scheme: tuples enter the eddy with low priority, and when they are returned to the eddy from an operator they are given high priority. This simple priority scheme ensures that tuples flow completely through the eddy before new tuples are consumed from the inputs, ensuring that the eddy does not become "clogged" with new tuples

Question 2.19. How is an eddy's tuple buffer implemented? Describe the tuple priority scheme and how the naive eddy can "learn" to evaluate lower-cost operators first.

A naive eddy works fairly well, actually. Consider a select with 2 expensive predicates "and"-ed together. If we pipeline the two predicates, we should run the faster one first. An eddy will learn to evaluate the faster one first because of "back pressure" — the lower cost operator will consume tuples more quickly, increasing the priority on their tuples.

The naive eddy works well when selectivity is kept constant. But what if selectivity also changes? Naive eddies perform about halfway between best and worst plans. To solve this dilemma, they change the priority scheme to favor operators based on consumption *and* production rate. Note that consumption (input) rate of an operator is determined by cost alone, but production (output) rate is determined by a product of cost and selectivity. To track consumption and production over time, they enhanced the priority scheme with lottery scheduling (an algorithm to ”learn” the selectivity). The eddy gives a ticket to an operator when it gives the operator a tuple, and it takes the ticket back when the operator returns a tuple. Eddies give tuples to operators proportional to the number of tickets they have. The lottery scheduling algorithm works well on joins too, without fancy selectivity estimation.

Question 2.20. Explain the lottery scheduling algorithm that an eddy can use to route tuples to operators with lower selectivity factors.

2.4.4 Responding to Dynamic Fluctuations

Lottery scheduling favors all tickets equally. As a result, an operator that earns many tickets early in a query may become so wealthy that it will take a great deal of time for it to lose ground to the top achievers in recent history. But more recent tickets should be favored over older tickets. To solve this, we use windows.

The eddy keeps track of two counts for each operator: a number of banked tickets, and a number of escrow tickets. Banked tickets are used when running a lottery. Escrow tickets are used to measure efficiency during the window. At the beginning of the window, the value of the escrow account replaces the value of the banked account, and the escrow account is reset. This scheme ensures that operators “re-prove themselves” each window.

2.4.5 Advantages and Disadvantages of Eddies

There are a number of advantages of eddies:

- Eddies allow for bushy joins
- No need to have great selectivity estimates
- Eddies can optimize for time instead of just I/O
- Eddies can dynamically reorder joins

Some disadvantages are:

- If the join order for the user's query is forced, eddies don't really help
- Depending on the workload, the overhead of dynamic routing and priority queues might not be worth it. So eddies will have worse best-case performance than a fixed query plan

Question 2.21. Describe a strength and weakness of eddies.

2.5 Worst-Case Optimal Join Algorithms [42]

Efficient join processing is one of the most fundamental and well-studied tasks in database research. In this work, we examine algorithms for natural join queries over many relations and describe a novel algorithm to process these queries optimally in terms of worst-case data complexity. Our result builds on recent work by Atserias, Grohe, and Marx, who gave bounds on the size of a full conjunctive query in terms of the sizes of the individual relations in the body of the query. These bounds, however, are not constructive: they rely on Shearer's entropy inequality which is information-theoretic. Thus, the previous results leave open the question of whether there exist algorithms whose running time achieve these optimal bounds. An answer to this question may be interesting to database practice, as we show in this paper that any project-join plan is polynomially slower than the optimal bound for some queries. We construct an algorithm whose running time is worst case optimal for all natural join queries. Our result may be of independent interest, as our algorithm also yields a constructive proof of the general fractional cover bound by Atserias, Grohe, and Marx without using Shearer's inequality. In addition, we show that this bound is equivalent to a geometric inequality by Bollobás and Thomason, one of whose special cases is the famous Loomis-Whitney inequality. Hence, our results algorithmically prove these inequalities as well. Finally, we discuss how our algorithm can be used to compute a relaxed notion of joins.

Previous work showed that, if you have a full conjunctive query (where every variable in the body appears in the head), you can tightly bound the number of resulting tuples—i.e., if you have relations R, S, and T where the cardinalities of each are N, a tight upper bound is $N^{3/2}$ number of resulting tuples in the join. This paper asks, can we construct an algorithm that will *run* in $O(N^{3/2})$ time?

Definition 2.3. Optimal Worst-case Join Evaluation Problem (Optimal Join Problem). Given a fixed database schema $\bar{R} = \{R_i(\bar{A}_i)\}_{i=1}^m$ and an m-tuple of integers $\bar{N} = (N_1, \ldots, N_m)$. Let q be the natural join query joining the relations in $\bar{R}$ and let $I(\bar{N})$ be the set of all instances such that $|R_i^I| = N_i$ for $i = 1, \ldots, m$. Define $U = \sup_{I \in I(\bar{N})} |q(I)|$. Then the optimal worst case join evaluation problem is to evaluate q in time $O(U + \sum_{i=1}^m N_i)$.

Given a natural join query like $R(A, B), S(B, C), T(C, A)$, we construct a graph as follows:

- Each attribute A, B, C is a vertex
- Each relation becomes an edge. For example $R(A, B)$ becomes the edge $\{A, B\}$

Assign a non-negative weight to each edge. Go to each vertex and sum up the weights assigned to its corresponding edges. Make sure that this sum is greater than or equal to 1 for every vertex. The number of tuples in the output of this query is guaranteed to be less than $|R|^{\text{weight of } R} \times |S|^{\text{weight of } S} \times |T|^{\text{weight of } T}$ for any such edge cover. I don't understand the proof, but a worst case optimal join algorithm will always run in $O(\text{worst case size of } q \times \text{ sum of the sizes of all input relations})$.

In practice, worst case optimal isn't always what we're aiming for. Also, more standard join algorithms might be easier to make fast (e.g. cache locality, prefetching, distribution, etc.) Supposedly an example of a worst case optimal join algorithm is leapfrog join. [Shreya: What is this?]

2.6 Datalog and Recursive Query Processing—Sections 1-3 and 6 [28]

Abstract

In recent years, we have witnessed a revival of the use of recursive queries in a variety of emerging application domains such as data integration and exchange, information extraction, networking, and program analysis. A popular language used for expressing these queries is Datalog. This paper surveys for a general audience the Datalog language, recursive query processing, and optimization techniques. This survey differs from prior surveys written in the eighties and nineties in its comprehensiveness of topics, its coverage of recent developments and applications, and its emphasis on features and techniques beyond "classical" Datalog which are vital for practical applications. Specifically, the topics covered include the core Datalog language and various extensions, semantics, query optimizations, magic-sets optimizations, incremental view maintenance, aggregates, nega-

tion, and types. We conclude the paper with a survey of recent systems and applications that use Datalog and recursive queries.

2.6.1 Baby's First Example

Suppose we have the following Datalog program that computes whether there is a path from X to Y, given the function `link(X, Y)` if there is an edge from X to Y:

```
r1: reachable(X, Y) :- link(X, Y)
r2: reachable(X, Y) :- link(X, Z), reachable(Z, Y)
query(X,Y) :- reachable(X,Y)
```

Rule r1 expresses that X is reachable from Y if they are directly linked. Rule r2 is recursive: X is reachable from Y if there is a link from X to some Z where Z is reachable to Y.

Intuitively, rule evaluation can be understood as the repeated application of rules over existing tuples to derive new tuples, until no more new tuples can be derived (i.e. evaluation has reached a fixpoint). Each application of rules over existing tuples is referred to as an iteration. This evaluation strategy is often times referred to as the naive evaluation strategy. A semi-naive evaluation strategy does not re-evaluate rules we have already reached in a previous iteration.

Note that the above approach is a bottom-up evaluation technique, where existing facts are used as input to rule bodies to derive new facts. A fixpoint is reached when no new facts are derived. This is also known as a forward-chaining style of evaluation. Prolog is top-down, it starts from the query and expands the rule bodies in a top down fashion. Bottom up needs pruning, and Datalog can draw from a wealth of techniques.

2.6.2 Language and Semantics

A Datalog program is a collection of Datalog rules, each of which is of the form:

$$A :- B_1, B_2, \ldots, B_n$$

where $n \geq 0$, A is the head of the rule, and the conjunction of Bs is the body of the run. Some terminology:

- Term: constant or variable

- Atom: predicate symbol (function) with a list of terms as arguments (e.g., link(X, Y))
- Ground atom: only contains constant arguments
- Extensional database predicates (EDB predicates): correspond to source tables of the DB
- Intensional database predicates (IDB predicates): correspond to derived tables computed by Datalog programs
- Base atom: entity in the source table
- Derived atom: entity in derived table
- Database instance: set of ground atoms
- Source database instance: database containing only EDB atoms
- Active domain: set of all constants in the database
- Range restriction property: every variable occurring in the head of a rule must occur in a predicate atom in the body or be equivalent to a variable occurring in a predicate atom in the body
- Herbrand base of a collection of rules P, denoted as $B(P)$: set of all ground atoms that can be formed by the EDB or IDB predicates in P
- Ground instance: substitute all variables of the rule with constants

I am honestly not learning about the semantics here, but Datalog has three semantics: model-theoretic, least fixpoint, and proof-theoretic. Datalog with negation has two semantics: semipositive and stratified. There are some optimizations to improve the performance of aggregation to avoid redundant computations (e.g. for a shortest path query).

Datalog has found uses in the areas of program analysis (e.g. points-to analysis), declarative networking (e.g. NDLog and Bloom), data integration and exchange (e.g. schema mapping stuff), enterprise software (e.g. LogicBlox), concurrent programming, etc.

[Shreya: TODO: read more if time permits]

3 Transactions

In this section, we discuss the following papers:

- ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging
- Granularity of locks and degrees of consistency in a shared data base
- Concurrency Control in Distributed Database Systems
- Concurrency control performance modeling: alternatives and implications

3.1 Transaction Basics [45]

First, we will cover some basic concepts on transactions, summarized from the Cow Book on databases [45]. The ACID properties are as follows:

- Atomicity: either all actions are carried out, or none are carried out
- Consistency: each transaction must leave the DB in a "consistent" state, as defined by the user
- Isolation: transactions are isolated from other concurrently running transactions, so it is almost like transactions are run one at a time. The whole motivation for concurrency is to run I/O and CPU operations at the same time.
- Durability: if the system crashes, we can recover it to the state of all committed transactions

Concurrency techniques ensure consistency and isolation. Recovery techniques ensure durability and atomicity.

Definition 3.1. A **schedule** is a list of actions (read, write, commit, abort) from a set of transactions, and the order in which two actions of a transaction appear in a schedule must be the same order in which they appear in the transaction. If actions of different transactions are not interleaved (that is, transactions are executed start to finish, one by one, we call this schedule a **serial** schedule.

3.1.1 Serializability

Definition 3.2. A **serializable schedule** over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of committed transaction in S. In other words, even if operations within transactions are interleaved, it is effectively such that the transactions ran one at a time.

DBMSes may execute transactions in a way that is not equivalent to any serial execution. Two actions on the same data object **conflict** if at least one of them is a write. There are 3 main ways you can get anomalies due to interleaved execution—that is, partway through executing a schedule, you can get an inconsistent state:

- **Dirty read (WR conflicts):** $T1$ writes to object O, hasn't committed yet, and $T2$ reads from object O
- **Unrepeatable read (RW conflicts):** $T1$ reads O, then $T2$ modifies O and commits while $T1$ is still in progress
- **Overwriting committed data (WW conflicts):** $T1$ writes to O, then $T2$ also writes to O and commits, then $T1$ commits and the DB isn't updated with the latest value of O as defined by the commit timestamp of $T1$

Question 3.1. Discuss the 3 types of anomalies due to interleaved transactions, and give an example of each.

What happens when you need to abort transactions? You can *cascade* dependency aborts if the dependencies have not been committed yet. But if the dependency has been committed, we can't undo it. We say that such a schedule is **unrecoverable**.

Definition 3.3. In a **recoverable** schedule, transactions commit if and only if all transactions whose changes they read commit. This avoids cascading aborts, which is nice.

On the flip side, if $T1$ aborts but dependent transaction $T2$ is still in progress and $T2$ overwrote an object written by $T1$, when the DBMS does the abort, the object will revert back to the value before $T1$—potentially undoing the effect of $T2$'s write. Strict two phase locking solves this problem (which we will discuss soon).

3.1.2 Lock-based Concurrency Control

DBMSes need to enforce that only serializable, recoverable schedules are allowed. They do this through locking. Different locking protocols use different types of locks.

2PL. If a transaction wants to read or write to an object, it requests a shared or exclusive lock on the object, respectively. After the first unlock operation, it requests no new locks. This gives you serializability.

Conservative 2PL. Here, transactions obtain all locks needed before they even begin operations. This is to ensure that a transaction that already holds some locks will not block waiting for other locks. Conservative 2PL prevents deadlocks.

Strict 2PL. The rules in strict 2PL are as follows: if a transaction wants to read or write to an object, it requests a shared or exclusive lock on the object, respectively. The transaction must first acquire all locks before executing operations in the transaction. Shared locks can be released whenever (after all locks have been acquired), but strict 2PL releases all exclusive locks after transactions commit. This guarantees cascadeless recovery.

Rigorous 2PL. This is the same as 2PL, except all locks are released only after transactions commit/abort, not just exclusive locks. There are no deadlocks in rigorous 2PL.

Unfortunately strict 2PL can still have deadlocks. The simplest way to detect deadlocks is to use a timeout mechanism. In practice, fewer than 1% of transactions are involved in deadlock, and there are relatively few aborts. So locking overhead primarily comes from blocking delays.

Throughput can be increased in three ways:

- Lock smallest size objects possible
- Reduce time transactions hold locks
- Reducing hot spots (objects frequently accessed and modified)

Locking the smallest size objects has some problems. For example, consider a query that returns the minimum age of a customer. If a transaction is trying to execute this query, it will acquire locks for each row of the table. In the middle of the transaction, another transaction might add a new row to the table, which gets a lock that the first transaction doesn't have and thus might impact the result of the query. This is called the **phantom problem**, because locking schemes don't (by themselves) impose order on the two transactions.

Question 3.2. Describe the phantom problem with an example.

3.1.3 Isolation Levels

Read-only access modes only require shared locks to be obtained, increasing concurrency. For the read-write access mode, you need an exclusive lock. The **isolation level** controls the extent to which a given transaction is exposed to the actions of the other transactions executing concurrently. There are 4 isolation levels:

- Read uncommited: can do dirty and unrepeatable reads. Has phantom problem. Achieved without locking.
- Read committed: cannot do dirty reads, but can do unrepeatable reads. Has phantom problem. This is achieved by read requests acquiring a read lock before accessing an object and unlocking it immediately after access. Write locks are held until the end.
- Repeatable read: no dirty reads, no unrepeatable reads. Still has phantom problem. This is achieved by read requests acquiring a read lock before accessing an object, and holding the lock until end-of-transaction. However, only individual objects are locked here, not sets of objects.
- Serializable: guaranteed full serializable access, no phantom problem. Strict 2PL (on sets of locks) is used here.

Question 3.3. Describe the 4 isolation levels. For each isolation level, how do you use locking to obtain it? Why, for the serializable isolation level, do we need 2PL on sets of locks, not just individual locks?

3.1.4 2PL, Serializability, and Recoverability

Here, we will discuss how locking protocols guarantee serializability and recoverability. First, we need a finer-grained version of serializability.

Definition 3.4. Two schedules are **conflict equivalent** if, for every pair of conflicting actions of two committed transactions, the pair is ordered the same way in the schedules. A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule. Every conflict serializable schedule is serializable if we assume that items in the database are neither added nor deleted.

To capture conflicts, we can use a serializability graph, constructed as follows:

- A node for each committed transaction
- An edge from T_i to T_j if an action of T_i precedes *and* conflicts with one of T_j's actions

A schedule is conflict serializable if and only if its graph is acyclic. 2PL will ensure the graph is acyclic. Strict schedules (enforced by strict 2PL) are recoverable and don't cause cascading aborts.

Conflict serializability is sufficient but not necessary for serializability.

Definition 3.5. Two schedules are **view equivalent** if:

1. If T_i reads the initial value of object A in $S1$, it also must read the initial value of A in $S2$
2. If T_i reads a value of A written by T_j in $S1$, it must also read the value of A written by T_j in $S2$
3. For each data object A, the transaction that performs the final write on A in $S1$ must perform the final write on $S2$

A schedule is **view serializable** if it is view equivalent to a serial schedule.

Every conflict serializable schedule is view serializable, but not vice versa. Enforcing view serializability is much harder than conflict serializability.

3.1.5 Lock Conversions and Deadlocks

Sometimes transactions might need to upgrade their locks—for example, in a SQL update statement with a WHERE clause, the transaction might initially hold shared locks on all the rows, then upgrade to exclusive locks for rows where the predicate is satisfied. The DBMS will immediately grant the upgrade request if no other transaction holds a shared lock, else it will put the transaction in the front of the queue for the lock. This helps reduce the chance of deadlock, but one class of deadlocks holds: if two transactions both hold shared locks and want the same upgrade.

One solution to avoid the need for lock upgrades is to use lock downgrades instead. The transaction can initially only obtain exclusive locks, then downgrade to a shared lock if possible. For example, it can obtain exclusive locks for all the rows, then when a row does not satisfy the predicate, it can downgrade to a shared lock. This does not violate 2PL as long as the transaction did not make any writes to the object locked.

Question 3.4. Discuss the relative merits of lock upgrades and downgrades.

Deadlock. In practice, DBMSes periodically check for deadlocks. The lock manager maintains a "waits-for" graph to detect deadlock cycles. The nodes correspond to active transactions, and there is an edge from T_i to T_j if and only if T_i is waiting for T_j to release a lock.

If a cycle is detected, the DBMS initiates an abort for a transaction that is on a cycle and releases its locks. Typically the one with the least locks is aborted (because it may have done the fewest operations and thus can undo efficiently), or one that was repeatedly restarted.

However, in some cases (high level of contention for locks), we may want a deadlock prevention scheme. The DBMS can give each transaction a timestamp when it starts up. Lower timestamps have higher priority. If a transaction T_i requests a lock that T_j is holding, there are two policies we could use:

- Wait-die: if T_i has higher priority, it is allowed to wait; else it will abort.
- Wound-wait: if T_i has higher priority, abort T_j; else wait.

When a transaction is aborted and restarted, in this scheme, the priority must be the same timestamp it was before abort. Otherwise transactions might infinitely repeatedly be aborted.

Conservative 2PL can also be used to detect deadlock (acquire all locks before doing any operation for a transaction). This reduces time locks are held only if lock contention is heavy. It has high overhead when lock contention is low because if a transaction fails to obtain just one of many locks, it has to release all the other locks too. It is not used in practice.

Question 3.5. Discuss the wait-die and wound-wait deadlock prevention mechanisms. List an advantage/disadvantage of each. Why are detection schemes more commonly used than prevention schemes?

3.1.6 Concurrency Control in B+ Trees

To solve the phantom problem, ideally we'd lock distinct predicates. However, predicate matching is a super hard problem, so it's not commonly used. A simple strategy is to treat every page in a B+ tree as a data object and give it its own lock. But this will lead to high resource contention in the root of the tree or higher levels.

There are two observations to give us better locking strategies for higher levels of trees:

1. Higher levels of trees are only direct searches; real data is in the leaf notes
2. If there's an insert in the tree, it only needs an exclusive lock if there will be a split

With respect to observation 1, we can obtain shared locks on nodes starting on a root and proceeding on a path to the desired leaf. When a child lock is obtained, a parent lock can be released (since searches don't go up a tree). For inserts, we use exclusive locks, but parent locks can only be released if the child node is not full (so an insert won't cause a split). This technique of locking a child and releasing a parent is called **lock-coupling** or **crabbing**.

This strategy is an improvement over naive 2PL, but still there are exclusive locks set unnecessarily, motivating multiple-granularity locking.

3.1.7 Multiple-Granularity Locking

Databases are inherently hierarchical. A DB contains several files, each file is a collection of pages, each page is a collection of records, etc. So transactions will need to access locks in a "hierarchical" fashion. How do you maintain good concurrency while minimizing locking overhead?

In multiple granularity locking, unlike B+ tree locking, when we lock a node, we implicitly lock all its children. In addition to shared and exclusive locks, we have **intention shared (IS)** and **intention exclusive (IX)** locks. To lock a node in S mode, a transaction must lock all its ancestors in IS mode. To lock a node in X mode, a transaction must lock all its ancestors in IX mode. This ensures that no other transaction holds a lock on an ancestor that conflicts with the required S or X lock on the node.

A common situation is that a transaction needs to read an entire file but modify only a few records. So it will need an S lock on the file and IX lock on a few of the objects so that it can subsequently lock contained objects in X mode. So they have another kind of lock, a **SIX** lock, that is equivalent to holding a S and IX lock. One key point is that locks must be released in leaf-to-root order.

Multiple granularity locking must be used with 2PL to guarantee serializability. The 2PL locking dictates when locks are released.

Question 3.6. Discuss the lock modes in multiple-granularity locking.

3.1.8 Optimistic Concurrency Control

Suppose we have a system with light contention for locks. Then the locking overhead might not be worth it. In optimistic concurrency control, the idea is to be as permissive as possible for transactions. Transactions have 3 phases:

1. Read: the transaction executes, reading values from the DB and writing to a private workspace
2. Validation: if the transaction wants to commit, the DBMS checks whether it might have conflicted with any of the other transactions. If it did, the transaction is aborted and the workspace is cleared
3. Write: if there were no conflicts, the changes in the private workspace are copied into the DB

A key question here is how to do the validation. Every transaction is assigned a timestamp at the beginning of validation, denoted TS. For every pair of transactions T_i and T_j such that $TS(T_i) < TS(T_j)$, one of the following validation conditions must hold:

- T_i completes all 3 phases before T_j begins any phase

- T_i completes all 3 phases before T_j started the write phase, and T_i did not write any object that T_j read
- T_i completes its read phase before T_j completed its read phase, and T_i does not write any object that is either read or written by T_j (prevents RW, WR, or WW conflicts)

Checking these validation criteria requires us to maintain lists of objects read and written by each transaction. Also while one transaction is being validated, no other transactions can commit. Also the write phase for a transaction must be completed before other transactions can be validated. Essentially validate + write are atomic.

The locking overheads in this scheme are replaced with recording read/write lists for transactions, checking conflicts, and copying private workspace changes.

Question 3.7. Describe the 3 phases of transactions in optimistic concurrency control, as well as the criteria for validation.

Question 3.8. What overheads exist in optimistic concurrency control?

Improved Conflict Resolution. OCC is unnecessarily conservative—namely, T_i cannot write any object read by T_j. In theory, there should be no problem if T_i's writes are performed before T_j reads them. But in OCC, we have no way to tell when T_i wrote objects relative to T_j reading them. So we will use finer-grained resolution for data conflicts. The basic idea is as follows: when T_i is committed, the DBMS checks whether its written items are read by any concurrent (yet to be validated) transaction. If there is such a transaction, it will fail in its validation—so we can kill it, or allow it to discover its failure when it's validated (kill vs die policy). It is implemented via a hash table: before reading a data item, a transaction T enters an access entry in a hash table, containing the transaction ID, data object ID, and a modified flag (initially set to false). Entries are hashed on the data object ID, and a temporary exclusive lock is obtained on the hash bucket while the data item is copied into the transaction's private workspace.

During validation of T, the hash buckets of all data objects accessed by T are again locked in exclusive mode to check if T has conflicts. T has encountered a conflict if the modified flag is set to true in one of its access entries (in the die policy). If T is successfully validated, we additionally lock all the hash buckets of objects modified by T, set the modified flag to true for each access entry for the object, and kill the corresponding transactions (in the kill policy). Then we complete T's write phase.

Why would we ever use the die policy instead of the kill policy? In the die policy, if a transaction finds that it needs to die during validation, its data might be prefetched into the DB and we can avoid recopying data from disk. Then its chances of validation will be higher when it completes again.

Question 3.9. Describe how OCC validation can be relaxed. What data structures do we need, and how do we prevent against conflicts?

3.1.9 Timestamp-Based Concurrency Control

In lock-based CC, the order of the locks obtained dictates serializability. In OCC, the timestamps, checked during validation, dictate serializability. How else can we use timestamps?

At execution time, if we assign a transaction a timestamp at startup, we must ensure that if a_i of T_i conflicts with a_j of T_j, a_i occurs before a_j if $TS(T_i) < TS(T_j)$. To implement this,we give every object O a read timestamp $RTS(O)$ and write timestamp $WTS(O)$. Then we have the following rules:

- If T wants to read object O, if $TS(T) < WTS(O)$, it aborts. Else, it performs the read and sets $RTS(O) = TS(T)$.
- If T wants to write object O, if $TS(T) < RTS(O)$, it aborts. Else, it checks if $TS(T) < WTS(O)$—if so, it might abort if we care for conflicting writes. But in practice, we don't (this is called the **Thomas Write Rule**). Then we write the object and set $WTS(O) = TS(T)$.

In the Thomas Write Rule, we do not get conflict serializability, because we can get conflicting writes.

Note that if T is restarted with the same timestamp, it will abort again! So when T restarts, we need to reset its timestamp. This is different from 2PL for deadlock prevention, where we must use the old timestamps.

One drawback is that every change to the RTS or WTS must be written to disk and recorded in the log for recovery. This is significant overhead.

Question 3.10. Suppose a transaction T is aborted. What timestamp will it get upon restart if we use a 2PL deadlock prevention mechanism? What about timestamp-based concurrency control?

Question 3.11. How can we get conflict serializability in timestamp-based concurrency control?

The timestamp protocol does not give us recoverable schedules (where transactions commit only after all transactions they read from commit), unfortunately. We can get recoverability by buffering all write actions until a transaction commits. This is considerable added overhead—so timestamp-based concurrency control doesn't win over 2PL in centralized systems, at least. It's typically used in distributed systems.

Question 3.12. What are sources of overhead in timestamp-based concurrency control?

3.1.10 Multiversion Concurrency Control

This is similar to timestamp concurrency control, except we maintain several versions of a database object so reads are never blocked. Each version of an object has an associated write timestamp. When a transaction T_i wants to read an object, it reads the most recent version (max timestamp) whose write timestamp precedes $TS(T_i)$.

Each object has one read timestamp, while its versions each have a write timestamp. The rules are as follows:

- If T_i reads O, it sets O's read timestamp to $\max(TS(T_i), RTS(O))$ and reads the most recent version with $WTS(O) < TS(T_i)$
- If T_i writes to O: if $TS(T_i) < RTS(O)$, abort T_i and restart it with a new, larger timestamp. Else, create a new version of O and set the version's write timestamp and object's read timestamp to $TS(T_i)$

The drawbacks of this approach are similar to time-based concurrency control, plus the overhead of maintaining versions.

Question 3.13. What is a strength of multi-version concurrency control? Drawback?

Question 3.14. Compare lock-based, optimistic, timestamp-based, and multiversion concurrency control techniques.

3.2 ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging [41]

Abstract

In this paper we present a simple and efficient method, called ARIES (Algorithm for Recouery and Isolation Exploiting Semantics), which supports partial rollbacks of transactions, finegranularity (e. g., record) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of repeating history to redo all missing updates before performing the rollbacks of the loser transactions during restart after a system failure. ARIES uses a log sequence number in each page to correlate the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate chaining of the log records written during rollbacks to those written during forward progress, a bounded amount of logging is ensured during rollbacks even in the face of repeated failures during restart or of nested rollbacks We deal with a variety of features that are very Important in building and operating an industrial-strength transaction processing system ARIES supports fuzzy checkpoints, selective and deferred restart, fuzzy image copies, media recovery, and high concurrency lock modes (e. g., increment /decrement) which exploit the semantics of the operations and require the ability to perform operation logging. ARIES is flexible with respect to the kinds of buffer management policies that can be implemented. It supports objects of varying length efficiently. By enabling parallelism during restart, page-oriented redo, and logical undo, it enhances concurrency and performance. We show why some of the System R paradigms for logging and recovery, which were based on the shadow page technique, need to be changed in the context of WAL. We compare ARIES to the WAL-based recovery methods of DB2, IMS, and Tandem systems. ARIES is applicable not only to database management systems but also to persistent object-oriented languages, recoverable file systems and transaction-based operating systems. ARIES has been implemented, to varying degrees, in IBM's OS/2TM Extended Edition Database Manager, DB2, Workstation Data Save Facility/VM, Starburst and QuickSilver, and in the University of Wisconsin's EXODUS and Gamma database machine

Refer to this https://www.shreya-shankar.com/aries/ for more information on ARIES.

The recovery manager of a DBMS ensures atomicity and durability, in the face of system crashes and media failures. This is one of the hardest components of a DBMS to implement. ARIES has 3 phases:

1. Analysis: identify dirty pages in the buffer pool at the time of crash (pages that haven't yet been written to disk)
2. Redo: repeat actions to get the DB to the state it was at the time of the crash
3. Undo: abort all "in-progress" transactions (ones that didn't commit) so the DB only reflects the actions of committed transactions

ARIES is a **steal, no-force** approach to recovery. This means transactions can "steal" pages in the buffer from other uncommitted transactions, and when they have finished making writes, the writes can be pushed to disk on their own time (no-force).

There are 3 principles behind ARIES:

- Write-ahead logging: any change to an object in the DB is recorded in a log (which is written to stable storage) before the change is written to disk
- Replay history during redo: on restart following crash, ARIES will retrace all actions of the DBMS
- Logging changes during undo: while undoing a transaction, we log the abort-related changes made such that actions are not repeated in the event of repeating restarts

3.2.1 The Log

The log, a file of records, is periodically forced to stable storage. The most recent portion of the log, or the **log tail**, is kept in main memory. Every log record has a unique ID (**log sequence number, or LSN**). LSNs are assigned in monotonically increasing order. The **flushedLSN** is the LSN where the log was last forced to disk.

For recovery, every page in the database has a **pageLSN**, or the LSN corresponding to the log record that most recently updated it. A log record is written for each of the following actions:

- Page update: append UPDATE record with new and old data, change pageLSN. Page must be pinned in buffer while update operations are being carried out.

- Commit: append COMMIT record, force log to disk, refresh flushedLSN, clean up transaction from main memory
- Abort: append ABORT record, initiate undo
- End: after transaction is cleaned up from main memory, append END record
- Undo: undo the update, append CLR (compensation log record) record

Every log record has the following fields:

- LSN
- prevLSN (previous LSN for the transaction, used for undo)
- transID (transaction ID)
- type (record type)

CLRs also have another field—**undoNextLSN**—that tells us the LSN of the next record to be undone. The CLR describes an action that will never be undone—it contains the information needed to redo the change described, but not reverse it.

3.2.2 Other Recovery-Related Structures

Transaction Table. This table keeps one entry per active transaction containing the following metadata: transaction ID, **lastLSN** (most recent log record for this transaction, in case we need to undo), and **status** (in-progress, committed, aborted).

Dirty Page Table. This table keeps an entry for every dirty page in the buffer pool. Each entry contains the page ID and the **recLSN**, or the LSN of the first log record that dirtied the page (useful for redo).

These tables are maintained by the transaction manager and the buffer manager. During a crash, these tables are reconstructed in the analysis phase.

WAL. Write-ahead logging ensures recovery. The definition of a committed transaction is one where all its log records, including the COMMIT record, have been written to stable storage. Even in this no-force approach, the log tail is forced to stable storage upon commit. In the force approach, however, all pages modified by the transaction (including the log tail) are forced to disk upon commit.

Question 3.15. Discuss what write-ahead logging (WAL) means, and whether or not any changes are forced to disk upon commit. Compare the force & no-force approaches.

3.2.3 Checkpointing

A checkpoint is a snapshot of the DBMS state. It reduces the amount of work we have to do for recovery after a crash. Checkpointing has 3 steps: first, a `BEGIN_CHECKPOINT` record gets written to the log. Then, the current contents of transaction and dirty page tables get written to the log, along with an `END_CHECKPOINT` record. Finally, a special master record containing the LSN of the `BEGIN_CHECKPOINT` record gets written to stable storage. The only guarantee we have here is the transaction and dirty page tables at the time of the `BEGIN_CHECKPOINT` record.

This is not too expensive because it doesn't involve writing out pages in the buffer pool. The effectiveness of checkpointing though is limited to the minimum recLSN of pages in the dirty page table (because we use this for redoing). So it might help to periodically write dirty pages to disk via a background process.

Note that checkpoints primarily save time in the analysis phase, not redo or undo. This is because we anyways have to redo starting from the minimum recLSN, and undo from the maximum lastLSN.

3.2.4 Executing the Algorithm

In analysis, we start from the `BEGIN_CHECKPOINT` and reconstruct the dirty page and transaction tables. Note that the dirty page table, after reconstruction, may have pages that were written to disk. That is ok. The overhead of logging a `END_WRITE` log record at completion of each write operation is not worth it.

In redo, we re-execute all redoable records (UPDATE or CLR) provided none of the following hold:

- The affected page is not in the dirty page table (meaning the change was already flushed to disk)
- The recLSN for the page in the dirty page table is greater than the LSN for the change (meaning the change was already flushed to disk)
- The pageLSN is greater than or equal to the LSN for the change (meaning the change was already flushed to disk)

Note 3.1. When we redo a change, we do not create an additional log record. Instead, we set the pageLSN to the LSN of the redone record.

In undo, we find all transactions in progress at the time of crash and abort them, using the lastLSN. Consider the set of lastLSNs we need to undo. The undo algorithm repeatedly chooses the largest LSN to undo in the set, adding prevLSNs of undone operations to the same set. The use of CLRs makes sure that undo actions are not performed multiple times for the same log record.

3.2.5 Interaction with Concurrency Control

Thanks to the repeating history paradigm and the use of CLRs, ARIES supports fine-granularity locks and logging of logical operations rather than just byte-level modifications.

Logging logical operations yields higher concurrency, even though the use of finer locks can lead to increased locking overhead. So there is a trade-off between different WAL schemes.

3.2.6 System R Recovery Algorithm

System R does not have logging or WAL. Transactions create a **shadow page** of the page they want to modify, and make the modification to the shadow page. Then the transaction rewires the page table pointer to the shadow page, even though other transactions see the original page table until the transaction commits. On commit, discard the original data pages in favor of shadow pages and make the shadow page table the official page table. Aborts are easy here—simply discard the shadow versions of the page table and pages.

This scheme is problematic. First, it becomes highly fragmented because of the shadow versions of pages being located far away from the original page. Second, concurrency is bad. Third, there is a huge overhead due to shadow pages. Fourth, aborts may run into deadlocks. So System R deprecated their recovery algorithm in favor of a WAL approach.

Question 3.16. Describe the ARIES algorithm (3 phases). What are the advantages of the ARIES approach?

Question 3.17. Name 3 types of log records and describe their functionality.

Question 3.18. Name 3 types of LSNs and describe their functionality.

Question 3.19. Explain what happens if there are crashes during the Undo phase. What is the role of CLRs? What if there are crashes during the Analysis and Redo phases?

Question 3.20. Record-level logging increases concurrency. What are the potential problems, and how does ARIES address them?

To answer the question above, transactions may perform logical operations that modify the physical layout (e.g., update a B+ tree). So undoing a page modification is challenging because, concurrently, some other transaction might have modified the object such that now the value is on a new page. Consequently, ARIES stores the logical operation to undo via use of CLRs, not just the byte-level modifications.

3.2.7 Parallelism

There are some ways to achieve optimizations in ARIES:

- Apply redo in parallel with 1 thread per *page* being redone
- Redo CLRs in parallel
- Partial rollbacks could be used for deadlocks instead of total rollbacks
- Deferred restart: acquire locks during redo and then open up the sytem for txns when we start undo. A txn will block on a page if it needs to be undone. [Shreya: I don't quite understand this]
- Checkpointing during recovery can help avoid redundant work on subsequent crashes

- Media recovery: Media snapshot taken as of the most recent checkpoint. To recover the media, we reload it and start redo from the most recent checkpoint.
- Nested top action: sometimes we do not care for atomicity (e.g., extending a file). A top action is a sequence of things we don't want to undo. We log a dummy CLR at the end of the top action to prevent undo, whose UndoNxtLSN points to the next log record we're ok with undoing.

Question 3.21. What is a nested top action? Why is it useful? How is it logged?

Question 3.22. Why does ARIES assume 2PL?

3.3 Granularity of locks and degrees of consistency in a shared data base [27]

Abstract

The problem of choosing the appropriate granularity (size) of lockable objects is introduced and the tradeoff between concurrency and overhead is discusses. A locking protocol which allows simultaneous locking at various granularities by different transactions is presented. It is based on the introduction of additional lock modes besides the conventional share mode an5 exclusive mode. A proof is given of the equivalence of this protocol to a conventional one. Next the issue of consistency in a shared environment is analyze . This discussion is motivated by the realization that some existing data base systems use automatic lock protocols which insure protection only from certain types of inconsistencies (for instance those arising from transaction backup), thereby automatically providing a limited degree of consistency. Four degrees of consistency are introduced. They can be roughly characterized as follows: degree 0 protects others from your updates, degree I additionally provides protection from losing updates, degree 2 additionally provides protection from reading incorrect data items, and degree 3 additionally provides protection from reading incorrect relationships among data items (i.e. total protection). A discussion follows on the relationships of the four degrees to locking protocols, concurrency, overhead, recovery and transaction structure. Lastly, these ideas are compared with existing data management systems.

Concurrency is increased with finer granularity locks. But the locking overhead is also increased. What is a good tradeoff? How is it implemented?

3.3.1 Granularity of Locks

Lock modes are S, IS, X, IX, SIX. S and X implicitly lock the entire subtree. IS and IX locks imply that some descendant subtree might be locked in S or X mode. Lock nodes root to leaf; release locks leaf to root.

	X	S	IX	IS	SIX
X	n	n	n	n	n
S	n	y	n	y	n
IX	n	n	y	y	n
IS	n	y	y	y	y
SIX	n	n	n	y	n

Question 3.23. Why do we have SIX locks? What are alternatives to having SIX locks?

On request for a node:

- before requesting S or IS, all ancestors must be in IX or IS
- before requesting X, SIX, or IX, all ancestors must be held in SIX or IX mode
- locks should be released either at end of transaction (in any order), or in leaf to root order in the middle of a transaction

There is a DAG of locks, implying rules for lock access. To implicitly or explicitly lock a node, one should lock all parents of the node in the in the DAG with the appropriate mode.

A node is implicitly locked in shared mode if one ancestor is locked in shared mode. A node is implicitly locked in exclusive mode if all ancestors are locked in exclusive mode.

Question 3.24. What if both implicit shared and exclusive required a majority of ancestors? How would this work?

If the hierarchy/DAG is dynamically updated, we need a rule for changing parents of a node. Before moving a node in the lock graph, the node must be implicitly or explicitly granted X mode in both its old and its new position in the graph. Further, the node must not be moved in a way to introduce cycles. Alternatively, one can find a common ancestor of the old and new positions and get an X mode on it.

Question 3.25. How would you do hierarchical locking when nodes in the tree are moved around or new nodes are added?

For scheduling, maintain a FIFO queue of lock requests. Grant as many mutually compatiple requests from the top of the queue as possible. For conversions, sometimes we rerequest locks or try to upgrade locks. Only grant an upgrade when it is compatible with other outstanding locks. Prioritize conversions over new lock requests. Deadlock can occur, so you need a deadlock detection or prevention mechanism.

3.3.2 Degrees of Consistency

To cope with temporary inconsistencies, sequences of atomic actions are grouped to form transactions. There are four consistency guarantees, defined by interface, implementation, and proofs.

- **Degree 3:** no overwriting dirty data of other transactions, no committing writes until it completes all writes, no reading dirty data from other transactions, and other transactions do not dirty data read by current transaction before current one completes. This prevents non-repeatable reads (long X, long S).
- **Degree 2:** Same as degree 3 except other transactions can dirty data read by the current one before the current one completes. We can call this ”no dirty reads” and has (long X, short S).
- **Degree 1:** Same as degree 2 except can read dirty data from other transactions. We call this ”no dirty writes” and has long X. This is usually automatically provided. Recoverable but can read aborted stuff.
- **Degree 0:** Same as degree 1 except can commit writes before completing all writes. We call this ”no overwrite others’ dirty data” and has short X. These are unrecoverable.

Question 3.26. Discuss the 4 degrees of consistency. Describe lock holding times for each of S, X for each degree.

If all transactions run at least 1 degree consistent, system backup (undoing all in progress transactions) will not lose any update of completed transactions. If all transactions run at least 2 degree consistent, transaction backup (undoing any in progress transaction) will produce a consistent state.

3.3.3 Dependencies Among Transactions

- $T < T'$: if a is a W and a' is a W
- $T << T'$: if a is a W and a' is a W, or a is a W and a' is a R
- $T <<< T'$: if a is a W and a' is a W, or a is a W and a' is a R, or or a is a R and a' is a W

Assertion: a schedule (set of transactions over time) is degree 1 (2, or 3) consistent if and only if the relation $< *(<< *or <<< *)$ is a partial order. (Partial order: if $T < T'$, then we cannot have $T' < T$).

3.3.4 Backups and Recovery

Transactions are **recoverable** if they can be undone without undoing other transactions' updates. Transactions are **repeatable** if we can replay them and reproduce the original output. Recoverability requires system wide degree 1 consistency, repeatibility requires that all other transactions be at least degree 1 and that the repeatable transaction be degree 3.

Checkpoint and replay of at least degree 1 consistent transactions. Replaying degree 1 may give different results because degree 1 doesn't prevent reading of dirty data. Degree 2 guarantees consistency.

Question 3.27. What is recoverability? Repeatability? What degree consistency must all transactions run at to achieve recoverability? Repeatability? Consistency?

Degree 2 and degree 3 seem to have similar computational overhead which seems to be substantially larger than the overhead of degree 0 or 1 consistency.

3.4 Concurrency Control in Distributed Database Systems [8]

Abstract

In this paper we survey, consolidate, and present the state of the art in distributed database concurrency control. The heart of our analysts is a decomposition of the concurrency control problem into two major subproblems: read-write and write-write synchronization. We describe a series of synchronization techniques for solving each subproblem

and show how to combine these techniques into algorithms for solving the entire concurrency control problem. Such algorithms are called "concurrency control methods." We describe 48 principal methods, including all practical algorithms that have appeared in the literature plus several new ones. We concentrate on the structure and correctness of concurrency control algorithms. Issues of performance are given only secondary treatment.

The distributed database model is as follows: A distributed database management system (DDBMS) is a collection of sites interconnected by a network. Each site is a computer running one or both of the following software modules: a transaction manager (TM) or a data manager (DM). TMs supervise interactions between users and the DDBMS while DMs manage the actual database. TMs don't communicate with each other; DMs don't communicate with each other.

A database consists of a collection of logical data items, denoted X, Y, Z. A logical database state is an assignment of values to the logical data items composing a database. Each logical data item may be stored at any DM in the system or redundantly at several DMs. The stored copies of logical data item X are denoted $x_1 \ldots X_m$. We typically use x to denote an arbitrary stored data item. A stored database state is an assignment of values to the stored data items in a database.

A transaction T accesses the DBMS by issuing BEGIN, READ, WRITE, and END operations, which are processed as follows:

- BEGIN: The TM initializes for T a private workspace
- READ(X): The TM looks for a copy of X in T's private workspace. If the copy exists, its value is returned to T. Otherwise the TM issues dm-read(x) to the DM to retrieve a copy of X from the database, gives the retrieved value to T, and puts it into T's private workspace
- WRITE(X, new-value): The TM again checks the private workspace for a copy of X. If it finds one, the value is updated to new-value; otherwise a copy of X with the new value is created in the workspace. **The new value of X is not stored in the database at this time**
- END: The TM issues dm-write(x) for each logical data item X updated by T. Each dm-write(x) requests that the DM update the value of X in the stored database to the value of X in T's local workspace. When all dm-writes are processed, T is finished executing, and its private workspace is discarded

The "standard" implementation of atomic commitment is a procedure called two-phase commit.

Definition 3.6. Two-phase commit: Suppose T is updating data items X and Y. When T issues its END, the first phase of two-phase commit begins, during which the DM issues prewrite commands for X and Y. These commands instruct the DM to copy the values of X and Y from T's private workspace onto secure storage. If the DBMS fails during the first phase, no harm is done, since none of T's updates have yet been applied to the stored database. During the second phase, the TM issues dm-write commands for X and Y which instruct the DM to copy the values of X and Y into the stored database. If the DBMS fails during the second phase, the database may contain incorrect information, but since the values of X and Y are stored on secure storage, this inconsistency can be rectified when the system recovers.

Question 3.28. Describe the difference between pre-write commands and dm-write commands.

3.4.1 Decomposition of the Concurrency Control Problem

Two executions are computationally equivalent if (1) each dm-read operation reads data item values that were produced by the same dm-writes in both executions; and (2) the final dm-write on each data item is the same in both executions.

Concurrency control decomposed in terms of rw/wr conflicts, ww conflicts, and the glue that binds them together.

Definition 3.7. Execution E is serializable if (a) its rwr conflicts are acyclic, (b) its ww conflicts are acyclic, and (c) there is a total ordering of the transactions consistent with all rwr and ww conflicts.

3.4.2 Synchronization techniques based on 2PL

The basic way to implement 2PL in a distributed database is to distribute the schedulers along with the database, placing the scheduler for data item x at the DM where x is stored.

Writelocks are implicitly released by din-writes. However, to release readlocks, special lock-release operations are required. These lock releases may be transmitted in parallel

with the dinwrites, since the dm-writes signal the start of the shrinking phase. When a lock is released, the operations on the waiting queue of that data item are processed firstin/first-out (FIFO) order.

In the basic 2PL, we only need to get a read lock on the replica we're reading, but we need to get all replicas' write locks when writing.

Primary copy 2PL is a 2PL technique that pays attention to data redundancy. One copy of each logical data item is designated the primary copy; before accessing any copy of the logical data item, the appropriate lock must be obtained on the primary copy. For readlocks this technique requires more communication than basic 2PL. For writelocks, there is no extra communication—under basic 2PL, T would issue prewrites to all copies of X (thereby requesting writelocks on these data items) and then issue dm-writes to all copies. Under primary copy 2PL the same operations would be required, but only the prewrite (Xl) would request a writelock.

Voting 2PL: Suppose transaction T wants to write into X. Its TM sends prewrites to each DM holding a copy of X. For the voting protocol, the DM always responds immediately. It acknowledges receipt of the prewrite and says "lock set" or "lock blocked." (In the basic implementation it would not acknowledge at all until the lock is set.) Then the TM counts the number of lockset responses; if it's a majority, then the TM behaves as if all locks were set.

Why is this correct? Since only one transaction can hold a majority of locks on X at a time, only one transaction writing into X can be in its second commit phase at any time. All copies of X thereby have the same sequence of writes applied to them.

Voting 2PL is not appropriate for RW synchronization, since you only need one lock set for read (the copy that is read), not majority of locks.

Centralized 2PL: there is a central scheduler for 2PL. This has communication overhead.

Deadlock prevention. We can still use wound-wait and wait-die (with care not to assign a new timestamp). Care must be exercised in using preemptive deadlock prevention with two-phase commit: a transaction must not be aborted once the second phase of two-phase commit has begun. If a preemptive technique wishes to abort Tj, it checks with Tjs TM and cancels the abort if Tj has entered the second phase. No deadlock can result because if Tj is in the second phase, it cannot be waiting for any transactions. You can also do conservative 2PL (preordering of resources) to avoid deadlock altogether.

Deadlock detection. Implementing detection in a distributed database is difficult in that we must construct the waits-for graph efficiently. In the centralized approach, one site is designated the deadlock detector for the distributed system—every few minutes each scheduler sends its local waits-for graph to the deadlock detector, which unions the local graphs. In the hierarchical approach, the database sites are organized into a hierarchy (or

tree), with a deadlock detector at each node of the hierarchy. Deadlocks that are local to a single site are detected at that site; deadlocks involving two or more sites of the same region are detected by the regional deadlock detector; and so on. Hierarchical deadlock detection is based on the observation that more deadlocks are likely across closely related sites. The periodic nature of transmitting local graphs can cause phantom deadlock when a site goes down/crashes.

Definition 3.8. A transaction T may be restarted for reasons other than concurrency control (e.g., its site crashed). Until T's restart propagates to the deadlock detector, the deadlock detector can find a cycle in the waits-for graph that includes T. Such a cycle is called a **phantom deadlock**.

Question 3.29. Describe the 4 extensions of 2PL to distributed systems. Discuss pros and cons to each.

Question 3.30. What is a phantom deadlock? Why is it bad?

3.4.3 Synchronization techniques based on T/O

Basic Timestamp Ordering. Two-phase commit is incorporated by timestamping prewrites and accepting or rejecting prewrites instead of dm-writes. Once a scheduler S accepts a prewrite, it must guarantee to accept the corresponding dm-write no matter when the dm-write arrives. The effect is similar to setting a writelock on x for the duration of two-phase commit. To implement the above rules, S buffers dm-reads, dm-writes, and prewrites. With the Thomas Write Rule, we can ignore obsolete writes (if a write has an older timestamp than the thing it's writing, ignore the write), removing the need to incorporate 2PC into the ww synchronization algorithmm; the ww scheduler always accepts prewrites and never buffers dm-writes [Shreya: Don't understand this]

- Every transaction is given a timestamp, and operations are executed in timestamp order. A transaction is aborted (and restarted with a larger timestamp) if it detects an error in the timestamp ordering.
- With private workspaces and 2PC, we cannot allow a dm-read or dm-write until the prewritten writes with lower timestamps are done. Otherwise, for example, a read could read the wrong value.

- Buffer reads, prewrites, and writes. Reads cannot be issued until all previous pending prewrites are done. Writes cannot be issued until all previous pending reads are done.

Multiversion Timestamp Ordering.

Maintain list of read timestamps and list of (write timestamp, write value). Reads are never rejected and read the latest version. Writes are accepted only if no read occurred after it and before the next write As with basic T/O, we have to buffer reads, prewrites, and writes to ensure that a reads don't read the wrong value.

Suppose there is an out of order dm-write. That is, some dm-read(x) with higher timestamp arrived before this. Since W was not rejected, that means there was an intervening write after W and before the dm-read(x) in the schedule. So W had no effect on that read.

Let R be an out-of-order dm-read(x). That is, some dm-write(x) with higher timestamp arrived before this. R will ignore all writes greater than ts(R), so will read the same data it would have in the serial execution.

There are no rejected reads in rw, and there are no rejected writes in ww—so there is no need for 2 phase commit.

Conservative Timestamp Ordering.

This eliminates restarts during T/O scheduling. All TMs send dm-read and dm-write requests to DMs in increasing timestamp order. A dm-read is not executed until the min dm-write request from all TMs is bigger; a dm-write is not executed until the min dm-read request from all TMs is bigger.

We give each TM a transaction class: a read and write set. A transaction can execute at a TM only if its (predeclared) read and write sets are subsets. A DM only waits for timestamps from transaction classes with intersecting write sets (for ww conflicts) or intersecting read/write sets for (rw conflicts).

This is overly conservative; we serialize all operations, not just the conflicting ones. The T/O scheme also suffers from two other major problems: (1) If some TM never sends an operation to some scheduler, the scheduler will "get stuck" and stop outputting. To solve this problem, TMs are required to periodically send timestamped null operations to each scheduler in the absence of "real" traffic. A null operation is a dm-read or dm-write whose sole purpose is to convey timestamp information and thereby unblock "real" dm-reads and prewrites. (2) To avoid the first problem, every TM must communicate regularly with every scheduler; this is infeasible in large networks.

Timestamp management is a common criticism of T/O schedulers.

Question 3.31. Explain the timestamp ordering methods (basic, multiversion, conservative). Compare and contrast them.

3.4.4 Integrated Concurrency Control Methods

We could combine techniques for rw synchronization and ww synchronization. We could assign timestamps at locked points. Every item has a lock time and a txn is assigned a time larger than all lock times. [Shreya: TODO: read about this more, it's the cross product of existing methods]

Question 3.32. What is a locked point in a transaction? What is this concept useful for?

3.5 Concurrency Control Performance Modeling: Alternatives and Implications [3]

Abstract

A number of recent studies have examined the performance of concurrency control algorithms for database management systems. The results reported to date, rather than being definitive, have tended to be contradictory. In this paper, rather than presenting "yet another algorithm performance study," we critically investigate the assumptions made in the models used in past studies and their implications. We employ a fairly complete model of a database environment for studying the relative performance of three different approaches to the concurrency control problem under a variety of modeling assumptions. The three approaches studied represent different extremes in how transaction conflicts are dealt with, and the assumptions addressed pertain to the nature of the database system's resources, how transaction restarts are modeled, and the amount of information available to the concurrency control algorithm about transactions' reference strings. We show that differences in the underlying assumptions explain the seemingly contradictory performance results. We also address the question of how realistic the various assumptions are for actual database systems.

Performance studies are informative, but results have been contradictory. For example, Stonebraker and DeWitt suggest that algorithms that use blocking instead of restarts lead

to better performance, but others think that restarting leads to better performance than blocking.

In their model, they include users, physical resources, and characteristics of the workload. Differences in these result in seemingly contradictory results. In the study, they examine 3 concurrency control algorithms—blocking with deadlock detection, immediate restart, and optimistic (read, validate, write).

- Blocking: Read and write locks are acquired. Whenever a transaction is blocked, deadlock detection is run, and the youngest transaction is aborted if there is deadlock.
- Immediate-Restart: Read and write locks are acquired. Whenever a transaction cannot acquire a lock, it is aborted and restarted after some adjustable amount of time.
- Optimistic: Transactions proceed in three phases: read, validate, write. At the beginning of the validate step, they are assigned a timestamp, and transactions are aborted if they do not conform to the timestamp-prescribed serial ordering.

3.5.1 Performance Model

There are 3 main parts: the system model (hardware, DB size/granularity, etc), the user model (users arrive in batch or interactive via terminals), and the transaction model (workload characteristics).

Performance model includes a queueing system in which transactions are queued in a ready queue (waiting to become active), concurrency control queue (waiting to make concurrency control requests), object queue (waiting to access database objects), blocked queue (waiting to become unblocked), or update queue (waiting to perform a deferred update).

Fixed model parameters included: number of objects in a DB, mean/min/max size of transaction (num objects it touches), probability of writing an item it read, think time, restart delay, number of terminals, multiprogramming level, number of cpus/disks, and the time for I/O and CPU actions.

There are other things the model does not capture, like degradation of systems under high pultiprogramming levels, page thrashing, context switching costs, etc.

3.5.2 Performance Metrics

Primary metric is the transaction throughput rate (num transactions completed per second). They also measured transaction delay, blocking ratio (number of times a transaction

blocks per commit), restart ratio (number of times a transaction restarts per commit), total disk utilization, useful disk utilization (fraction of disk time used to do work that was actually completed/not aborted).

3.5.3 Experiments

First 4 are resource related. Last 2 are transaction behavior related.

Experiment 1: Infinite Resources. The probablility of conflicts increases as the multiprogramming level increases. In blocking, more lock requests were denied and led to thrashing (blocking "compounds" other transactions not being able to proceed). In immediate restart, the throughput plateaued but didn't decrease—the restart delay effectively limits parallelism. This restart delay is dynamic, it is equal to the running average of the response time. Finally, optimistic algorithm performed best—aborted transactions are immediately replaced by new transactions, effectively keeping the multiprogramming level high. Optimistic scales logarithmically.

Experiment 2: Resource-Limited. All 3 algorithms thrash. Max throughput is obtained via blocking. Immediate-restart performed as good as or better than optimistic. There were more restarts with the optimistic algorithm, and each restart was more expensive; this is reflected in the relative useful disk utilizations for the two strategies. But even though immediate-restart seems better, it is only because it effectively limits parallelisms through the dynamic restart delay. If we apply this to the other algorithms, they will also perform well at high levels of multiprogramming.

Experiment 3: Multiple Resources. Blocking performs best up to 25 CPUs; then optimistic starts to win. When useful disk utilization is only about 30%, the system begins to behave like infinite resources.

Experiment 4: Interactive Workloads. Adding user think time makes it seem like there are infinite resources. Blocks for small think time, but optimistic performs best for larger think times.

Experiment 6: Modeling Restarts. Some analyses model a transaction restart as the spawning of a completely new transaction. These fake restarts lead to higher throughput (especially for Immediate-Restart and Optimistic) because they avoid repeated transaction conflict.

Experiment 7: Write Lock Acquisition. Some analyses have transactions acquire read-locks and then upgrade them to write-locks. Others have transactions immediately acquire write-locks if the object will ever be written to. Upgrading locks can lead to deadlock if two transactions concurrently write to the same object. For limited parallelism, this

doesn't have much of an effect. For higher degrees of parallelism, it can reduce throughput. With finite resources, blind write restarts transactions earlier (making restart look better).

Lessons learned:

- All 3 algos are the same with low contention for resources
- Blocking beats restarting, unless resource utilization is low
- With high think time, use optimistic
- False assumptions on restarts and lock upgrades made blocking look unnecessarily bad
- The dynamic delay helped prevent thrashing in immediate-restart

Question 3.33. When can upgrading locks lead to deadlock?

Question 3.34. What system utilization patterns warrant blocking? Immediate-restart or optimistic?

Question 3.35. Describe a way to prevent thrashing.

4 Indexing

Papers covered:

- Efficient locking for concurrent operations on B-trees
- Improved query performance with variant indexes
- The R*-tree: an efficient and robust access method for points and rectangles
- The log-structured merge-tree (LSM-tree)

4.1 Efficient Locking for Concurrent Operations on B-Trees [37]

Abstract

The B-tree and its variants have been found to be highly useful (both theoretically and in practice) for storing large amounts of information, especially on secondary storage devices. We examine the problem of overcoming the inherent difficulty of concurrent operations on such structures, using a practical storage model. A single additional "link" pointer in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Our solution compares favorably with earlier solutions in that the locking scheme is simpler (no read-locks are used) and only a (small) constant number of nodes are locked by any update process at any given time. An informal correctness proof for our system is given,

This paper introduces a locking protocol where each process only uses a small, constant number of locks at any time. Also, no search through the tree is ever prevented from reading any node (locks only prevent multiple update access).

4.1.1 Storage Model

Every node of the tree is stored on disk. We assume that some locking discipline is imposed on lock requests, for example, a FIFO discipline or locking administration by a supervisory process. There are 4 operations threads can perform: lock(x), unlock(x), get(x), and put(A, x). Get and put are atomic.

The definition of a B* tree is as follows:

1. Each path from the root to any leaf has the same length h

2. Each node except the root and the leaves has at least k + 1 sons. (k is a tree parameter; 2k is the maximum number of elements in a node)
3. The root is a leaf or has at least two sons
4. Each node has at most 2k + 1 sons
5. The keys for all of the data in the B*-tree are stored in the leaf nodes, which also contain pointers to the records of the database
6. Within each node, the keys are in ascending order
7. In the B*-tree an additional value, called the "high key," is sometimes appended to nonleaf nodes
8. If a leaf node has fewer than 2k entries, then a new entry and the pointer to the associated record are simply inserted into the node
9. f a leaf has 2k entries, then the new entry is inserted by splitting the node into two nodes, each with half of the entries from the old node. The new entry is inserted into one of these two nodes (in the appropriate position). Since one of the nodes is new, a new pointer must be inserted into the father of the old single node. The new pointer points to the new node; the new key is the key corresponding to the old half-node. In addition, the high key of each of the two new nodes is set

Question 4.1. What's the difference between B-tree and B* trees?

4.1.2 Problem with Concurrent B+ trees

Imagine one thread is searching for a value in the B+ tree and another thread is concurrently modifying the tree such that the value moves. The first thread will not be able to find the value!

The simplest protocol requires searches and insertions to lock every node along their path from root to leaf. This protocol is correct, but limits concurrency. Smarter protocols have insertions place write intention locks along a path and upgrade those locks to exclusive locks when performing a write. Searches can read nodes with write intention locks on them but not with exclusive locks on them.

Even smarter protocols lock a subsection of the tree and bubble this subsection upwards through the tree. The algorithm employs pioneer and follower locks, to prevent other processes from invading the region of the tree in which a particular process is performing modifications. B-link trees will do something similar but will guarantee that at most three nodes are locked at any given point in time.

4.1.3 B-link Trees

The B-link tree is a B*-tree modified by adding a single "link" pointer field to each node. This link field points to the next node at the same level of the tree as the current node, except that the link pointer of the rightmost node on a level is a null pointer. Also, each node has a "high key" — the last pointer.

For any given node in the tree (except the first node on any level) there are (usually) two pointers in the tree that point to that node (a "son" pointer from the father of the node and a link pointer from the left twin of the node). One of these pointers must be created first when a node is inserted into the tree. We specify that of these two, the link pointer must exist first; that is, it is legal to have a node in the tree that has no parent, but has a left twin.

Link pointers have the advantage that they are introduced simultaneously with the splitting of the node. Therefore, the link pointer serves as a "temporary fix" that allows correct concurrent operation, even before all of the usual tree pointers are changed for a new (split) node.

4.1.4 Search

If the search process examines a node and finds that the maximum value given by that node is less than u, then it infers some change has taken place in the current node that had not been indicated in the father at the time the father was examined by the search. The current node must have been split into two (or more) new nodes. The search must then rectify the error in its position in the tree by following the link pointer of the newly split node instead of by following a son pointer as it would ordinarily do. This procedure does no locking of any kind.

4.1.5 Insertion

Beginning at the root, we scan down the tree to the leaf node that should contain the value v. We also keep track of the rightmost node that we examined at each level during the descent through the tree. If we need to split a node, we replace the split node a with nodes a' and b' (with addition of v). We then proceed back up the tree (using our "remembered" list of nodes through which we searched) to insert entries for the new node (b') and for the new high key of a' in the parent of the leaf node. We flush b' to disk before a'.

At worst, we hold three locks at a time (infrequently even, only when it is necessary to follow a link pointer while inserting a pointer to a split node). In backing up the tree, in order to insert a pointer to the split half of the new node, the inserter finds that the old father of the split node is no longer the correct place to perform the insertion and

begins chaining across the level of nodes containing the father in order to find the correct insertion position for the pointer. Three nodes are locked only for the duration of one operation.

Deadlock freedom is guaranteed by the well-ordering of the locking scheme. Note the possibility that-as we backtrack up the tree-due to node splitting the node into which we must insert the new pointer may not be the same as that through which we passed on the way to the leaf. Rather, the old node we used during the descent through the tree may have been split; the correct insertion position is now in some node to the right of the one where we expected to insert the pointer.

4.1.6 Deletion

A simple way of handling deletions is to allow fewer than K entries in a leaf node. In order, then, to delete an entry from a leaf node, we perform operations on that node quite similar to those described above for case 1 of insertion. In particular, we perform a search for the node in which u should lie. We lock this node, read it into memory, and rewrite the node after removing the value u from the copy in primary memory.

In situations where excessive deletions cause the storage utilization of tree nodes to be unacceptably low, a batch reorganization or an underflow operation which locks the entire tree can be performed.

Question 4.2. Can we run into livelock in a B-link tree? How?

Question 4.3. Discuss locking efficiency in a B-link tree.

4.2 Improved Query Performance with Variant Indexes [43]

Abstract

The read-mostly environment of data warehousing makes it possible to use more complex indexes to speed up queries than in situations where concurrent updates are present. The current paper presents a short review of current indexing technology, including row-set representation by Bitmaps, and then introduces two approaches we call Bit-Sliced indexing and Projection indexing. A Projection index materializes all values of a column in

RID order, and a Bit-Sliced index essentially takes an orthogonal bit-by-bit view of the same data. While some of these concepts started with the MODEL 204 product, and both Bit-Sliced and Projection indexing are now fully realized in Sybase IQ, this is the first rigorous examination of such indexing capabilities in the literature. We compare algorithms that become feasible with these variant index types against algorithms using more conventional indexes. The analysis demonstrates important performance advantages for variant indexes in some types of SQL aggregation, predicate evaluation, and grouping. The paper concludes by introducing a new method whereby multi-dimensional group-by queries, reminiscent of OLAP/Datacube queries but with more flexibility, can be very efficiently performed.

B+ trees are not very good for read-heavy workloads, small number of unique values, his paper surveys different indexing techniques for OLAP-style queries. OLAP query performance depends on creating a set of summary tables to efficiently evaluate an expected set of queries. The summary tables pre-materialize needed aggregates, an approach that is possible only when the expected set of queries is known in advance. Specifically, the OLAP approach addresses queries that group by different combinations of columns, known as dimensions.

4.2.1 Indexing Definitions

- Value-list index: B+ tree that maps a key to a set of RIDs that have the same value. If rids are of the form (page id, offset), then the rid lists in the leaves of a B+ tree can be grouped by page id. The rids within a page's group do not need to repeat the page id. This saves space.
- Bitmap index: these are more space-efficient. If RIDs are 32 bits each, the size of the leaf level of the B+ tree is 32n, where n is the number of rows. The leaf level of a bitmap index is nk bits (where k is the number of columns). Bitmap operations are faster than RID set operations so long as $k < 100$ (bitmaps are at least 1% full).
- Projection index: this is just the column of a relation stored in order of row number. A query that only reads one column will read fewer bytes from the projection than from the whole relation. This is standard column store stuff.
- Bit-sliced index: view each integer in a column as a bitstring to get a list of bitstrings, then slice by column in the list and store a bitstring for every column of bits. A bit-sliced index will have as many elements as the number of bits in one of the bitstrings.

4.2.2 Comparing Index types for Aggregate Evaluation

- Evaluating Single-Column Sum Aggregates: bit-sliced index is the smallest I/O and cheapest. Projection index is second best.
- Evaluating Other Column Aggregate Functions: sum and average are best with bit-sliced index, projection index is best for column-product (product of different columns), max and min (and median most of the time) is best with value-list
- Evaluating Range Predicates: value-list is good for narrow ranges; bit-sliced is best for wide ranges

Question 4.4. Why, for range predicates, would a value-list index outperform bit-sliced indexes for a narrow range?

4.2.3 Evaluating OLAP-style Queries (join a fact table with multiple dim tables)

Create join indexes to minimize cost of join. The join indexes store all possible values of join keys with dimension tables, then you can filter another relation (usually the fact table) by these join keys.

We assume Projection indexes exist on F (fact table) for each of the group-by columns (these are Join Indexes, since the group-by columns are on the Dimension tables), and also for all columns of F involved in aggregates. If instead we have a value-list index, we perform nested for loops over the indexes to compute the intersections of groupsets.

We can accelerate this for multiple group-by attributes using segmentation and clustering—rows of F are partitioned into Segments, and query evaluation is performed on a Segment at a time. Segmentation is most effective when the number of rows per Segment is the number of bits that will fit on a disk page. With this Segment size, we can read the bits in an index entry that correspond to a segment by performing a single disk I/O.

We can also create groupset indexes: the Groupset index value can be represented by a simple integer, which represents the starting position of the first 1-bit in the Groupset, and the ending position of that Bitmap can be determined as one less than the starting position for the following index entry. This is good for multidimensional groupbys.

4.3 The R*-tree: an Efficient and Robust Access Method for Points and Rectangles [7]

Abstract

The R-tree, one of the most popular access methods for rectangles, is based on the heuristlc optimization of the area of the enclosing rectangle m each inner node. By running numerous experiments in a standardized testbed under highly varying data, queries and operations, we were able to design the R*-tree which incorporates a combined optimization of area, margin and overlap of each enclosing rectangle in the directory. Using our standardized testbed in an exhaustive performance comparison, it turned out that the R*-tree clearly outperforms the existing R-tree variants Guttman's linear and quadratic R-tree and Greene's variant of the R-tree. This superiority of the R*-tree holds for different types of queries and operations, such as map overlay. for both rectangles and multldimensional points m all experiments. From a practical point of view the R*-tree is very attractive because of the following two reasons: 1 It efficiently supports point and spatial data at the same time, and 2 Its implementation cost is only slightly higher than that of other R-trees.

Spatial Access Methods (SAMs) are approximations of spatial objects by the minimum bounding rectangle with the sides of the rectangle parallel to the axes of the data space. The R tree is the most popular SAM. An R-tree is a map from rectangles (or points) to values. You can insert a rectangle and corresponding value and then search for regions that contain a point, regions that intersect a rectangle, or regions that enclose a rectangle. It is stored similar to a B+ tree: search keys are bounding boxes. The box at a non-leaf node N is the smallest box that contains all boxes associated with the child nodes. The search for a single point can lead us down several paths in the tree, so we must check all leaf nodes in the found paths.

R-trees can optimize for many things: minimize rectangle area, minimize overlap, minimize perimeter, maximize storage utilization.

4.3.1 R-tree Variants

R-tree variants differ in their implementations of chooseSubtree (for insert) and split functions. Guttman's original chooseSubtree, they select the child that needs the least area enlargement to include the new data. Resolve ties by choosing the child with the smallest area. This method minimizes the rectangle areas in the R-tree. The original split method aims to minimize the sum of areas of the two bounding boxes—an exact algorithm would take exponential time, but there are quadratic and linear approximations.

Quadratic approximation. Iterate through all pairs of rectangles A, B to determine A and

B that maximizes (total bounding box - A - B). Put A and B into different groups. Then for all other rectangles C, compute the area increase of the group whose covering rectangle will have to be enlarged least to accommodate it. Pick the C with the least area increase and add it to the corresponding group. Repeat until there is no more C.

Greene's split method selects seeds similarly. Then it determines the axis (x or y) along which the seeds are the most distant and divides entries in half along the axis.

Minimizing overlap using a good insertion algorithm is very important for good search performance. The R* tree introduces the concept of forced reinserts to reduce overlap: when a node overflows, rather than immediately split, we remove some number of entries and reinsert them into the tree. They also may try to minimize box perimeter rather than area.

For internal nodes that point to leaf nodes, R* chooseSubtree selects the child which increases overlap the least (sum of intersecting areas with each other entry in the node). Ties are broken by least area increase and then least area. For internal nodes that point to other internal nodes, the R*-tree chooseSubtree method uses the original method (the child which requires the least area enlargement, ties broken by least area).

The R* split method sorts each entry along each axis (first by low point, ties broken by high point). Then we consider all possible splits of entries such that both partitions contain at least m entries. The axis which minimizes the sum of the perimeter of the bounding boxes for each partition is chosen. Next, the points are sorted along the chosen axis and the partition which minimizes bounding box overlap is selected (ties broken by minimum area).

Question 4.5. Describe R-tree chooseSubtree and split methods. Describe the variant of Greene's split method.

Question 4.6. What is the difference between an R* tree and an R-tree, in terms of the chooseSubtree method? Similarity?

Question 4.7. Describe the forced reinsert algorithm in R* trees.

4.4 The Log-Structured Merge-Tree (LSM-Tree) [44]

Abstract

High-performance transaction system applications typically insert rows in a History table to provide an activity trace; at the same time the transaction system generates log records for purposes of system recovery. Both types of generated information can benefit from efficient indexing. An example in a well-known setting is the TPC-A benchmark application, modified to support efficient queries on the History for account activity for specific accounts. This requires an index by account-id on the fast-growing History table. Unfortunately, standard disk-based index structures such as the B-tree will effectively double the I/O cost of the transaction to maintain an index such as this in real time, increasing the total system cost up to fifty percent. Clearly a method for maintaining a real-time index at low cost is desirable. The Log-Structured Merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period. The LSM-tree uses an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort. During this process all index values are continuously accessible to retrievals (aside from very short locking periods), either through the memory component or one of the disk components. The algorithm has greatly reduced disk arm movements compared to a traditional access methods such as B-trees, and will improve costperformance in domains where disk arm costs for inserts with traditional access methods overwhelm storage media costs. The LSM-tree approach also generalizes to operations other than insert and delete. However, indexed finds requiring immediate response will lose I/O efficiency in some cases, so the LSM-tree is most useful in applications where index inserts are more common than finds that retrieve the entries. This seems to be a common property for History tables and log files, for example. The conclusions of Section 6 compare the hybrid use of memory and disk components in the LSM-tree access method with the commonly understood advantage of the hybrid method to buffer disk pages in memory.

Random writes into classic (disk-based, B-tree indexed) databases are expensive, requiring 4 random-access I/Os on average for each random key write. This is not too bad, unless you're doing only a ton of writes and nothing else. If your system gets more writes than reads, you should batch up writes and make them sequential, and slowly move these writes into a more structured form on disk in the background. The LSM-tree is a realization of this idea for B+-tree indexes.

4.4.1 The two component LSM-tree algorithm

The C_0 component is in-memory, and the larger C_1 component is on disk. The first component may not be a B+ tree, but the second one is a tightly packed B+ tree with every

layer of the tree stored contiguously. Whenever the C_0 tree as a result of an insert reaches a threshold size near the maximum allotted, an ongoing rolling merge process serves to delete some contiguous segment of entries from the C_0 tree and merge it into the C_1 tree on disk.

The merge process works as follows: read some C_1 leaves into memory, merge them with C_0 leaves, and then write back to C_1. When a split occurs, parents of leaves must also be written back to C_1. This is performant because sequential disk access is fast, and updates are applied in batch. It is an important efficiency consideration of the LSM-tree that when the rolling merge process on a particular level of the C1 tree passes through nodes at a relatively high rate, all reads and writes are in multi-page blocks. By eliminating seek time and rotational latency, we expect to gain a large advantage over random page I/O involved in normal B-tree entry insertion.

For searches, first read from C_0, then C_1, C_2, etc. If the search key is not a primary key, all components must be read. If the search key is a primary key, it can stop whenever the key is found.

When an indexed row is deleted, if a key value entry is not found in the appropriate position in the C0 tree, a delete node entry (i.e., "tombstone") can be placed in that position, also indexed by the key value, but noting an entry Row ID (RID) to delete. The actual delete is done in the rolling merge process. In the meantime, find requests must be filtered through delete node entries so as to avoid returning references to deleted records. This filtering is easily performed during the search for the relevant keyvalue, since the delete node entry will be located in the appropriate keyvalue position of an earlier component than the entry itself, and in many cases this filter will reduce the overhead of determining an entry is deleted.

Long-latency searches can also be integrated into the merge process, slowly accumulating the results as the merges take place.

Question 4.8. What are strengths of the LSM tree?

Question 4.9. Describe the tombstone delete process and how that leads to more efficient deletes.

4.4.2 The multi-component LSM-tree

The larger the C0 component in comparison to the C1 component, the more efficiency is gained in the merge. If we have a huge C1 and small C0 (data in general is really big), we want to make multiple components. This reads slow but writes are still fast because they are buffered.

The advantage of an LSM-tree of three components is that batching efficiency can be geometrically improved by choosing C1 to optimize the combined ratio of size between C0 and C1 and between C1 and C2. As a result, the size of the C0 memory component can be made much smaller in proportion to the total index, with a significant improvement in cost. You can derive the optimal sizes of different components to minimize total cost for memory and disk.

Question 4.10. Why have multiple components in an LSM-tree?

4.4.3 Concurrency Control in the LSM-tree

There are three types of conflicts:

1. A find operation should not access a node of a disk-based component at the same time that a different process performing a rolling merge is modifying the contents of the node
2. A find or insert into the C0 component should not access the same part of the tree that a different process is simultaneously altering to perform a rolling merge out to C1
3. Merging of different pairs of disk components

For disk-backed components, entries being modified by a rolling merge are locked in exclusive mode, and nodes being read are locked in shared mode. Locks are acquired at the granularity of nodes. The locking protocol for C0 can use the same scheme. After a node is processed as part of a merge, all write locks are released to allow another merge to potentially overtake it.

4.4.4 Recovery in the LSM-tree

To demonstrate recovery of the LSM-tree index, it is important that we carefully define the form of a checkpoint and demonstrate that we know where to start in the sequential

log file, and how to apply successive logs, so as to deterministically replicate updates to the index that need to be recovered. When a checkpoint is requested at time T0, we pause all new updates and merges. We complete all merge steps in operation, release node locks, and create a checkpoint with the following actions:

- Write C0 to disk, resume insertions (but no new merges)
- Flush all dirty memory buffered nodes of disk-based components
- Create special checkpoint log with LSN of last inserted indexed row at T0, disk addresses of roots of all components, location of all merge cursors in all components, and information for dynamic allocation of new multi-page blocks
- Flush checkpoint information to disk
- Resume regular merges

Use replay-based recovery techniques at the time of a crash.

5 DBMS Architectures Revisited

The following papers are covered:

- C-Store: A Column-Oriented DBMS
- Hekaton: SQL Server's Memory-Optimized OLTP Engine
- Calvin: Fast Distributed Transactions for Partitioned Database Systems
- Spanner: Google's Globally-Distributed Database
- Building Efficient Query Engines in a High-Level Language

5.1 C-Store: A Column-Oriented DBMS [50]

Abstract

This paper presents the design of a read-optimized relational DBMS that contrasts sharply with most current systems, which are write-optimized. Among the many differences in its design are: storage of data by column rather than by row, careful coding and packing of objects into storage including main memory during query processing, storing an overlapping collection of columnoriented projections, rather than the current fare of tables and indexes, a non-traditional implementation of transactions which includes high availability and snapshot isolation for read-only transactions, and the extensive use of bitmap indexes to complement B-tree structures. We present preliminary performance data on a subset of TPC-H and show that the system we are building, C-Store, is substantially faster than popular commercial products. Hence, the architecture looks very encouraging. o

Systems that query large amounts of data in ad-hoc should be read-optimized. With a column store architecture, a DBMS need only read the values of columns required for processing a given query, and can avoid bringing into memory irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a sizeable performance advantage. Many warehouse systems maintain two copies of their data because the cost of recovery via DBMS log processing on a very large data set is prohibitive. This option is rendered increasingly attractive by the declining cost per byte of disks.

C-Store does the following: combine in a single piece of system software, both a read-optimized column store and an update/insert-oriented writeable store, connected by a tuple mover. At the top level, there is a small Writeable Store (WS) component, which is architected to support high performance inserts and updates. There is also a much

larger component called the Read-optimized Store (RS), which is capable of supporting very large amounts of information. RS, as the name implies, is optimized for read and supports only a very restricted form of insert, namely the batch movement of records from WS to RS, a task that is performed by the tuple mover.

Question 5.1. Discuss the hybrid RS-WS and tuple mover architecture of C-Store.

5.1.1 Data Model

Whereas most row stores implement physical tables directly and then add various indexes to speed access, C-Store implements only projections. Specifically, a C-Store projection is anchored on a given logical table, T, and contains one or more attributes from this table. In addition, a projection can contain any number of other attributes from other tables, as long as there is a sequence of n:1 relationships from the anchor table to the table containing an attribute.

Projections are horizontally partitioned into segments across a cluster on their sort key. Each segment associates every data value of every column with a storage key, SK. Values from different columns in the same segment with matching storage keys belong to the same logical row. To reconstruct all of the records in a table T from its various projections, C-Store uses join indexes.

Join indexes are very expensive to maintain. Modifying a projection requires modifying all the join indexes pointing into or out of it. This is why they are good for read-only workloads.

Fault tolerance is achieved by horizontally partitioning projections with redundancy.

5.1.2 RS

The storage keys are implicit (the ith tuple has storage key i). Columns in RS are compressed based on their sort order and based on their number of distinct values. There are 4 types of encoding schemes based on the proportion of distinct values a column contains:

1. Self-order, few distinct values: store a dense B+ tree mapping v to (v, offset, count)
2. Foreign-ordered, few distinct values: bitmap per unique value. The column is a sequence of tuples (v, b) where b is a bitmap indicating the positions where v is stored

3. Self order, many distinct values: present every value in the column as a delta from the previous value, stored as B+ tree

4. Foreign-order, many distinct values: uncompressed with B+ tree

5.1.3 WS

In order to avoid writing two optimizers, WS is also a column store and implements the identical physical DBMS design as RS. Hence, the same projections and join indexes are present in WS. However, the storage representation is drastically different because WS must be efficiently updatable transactionally.

Projections are not compressed (WS is trivial in size compared to RS), and storage keys are stored explicitly. WS projections are horizontally range partitioned like RS projections so segments can be located.

When a tuple is inserted, it is assigned a storage key that is larger than all current WS storage keys.

There are 2 B-trees. One stores columns as (value, storage key) with a B-tree on the second field. The other stores sort keys of each projection (s/value, sk/sort key) with B-trees on the sort key. To perform searches using the sort key, one uses the latter B-tree to find the storage keys of interest, and then uses the former collection of Btrees to find the other fields in the record.

5.1.4 Updates and Transactions

Tuple are inserted into WS. To avoid requiring synchronization for assigned storage keys, nodes use a unique id plus a local counter to assign storage keys. The local counter is initialized to be bigger than the largest storage key in RS to ensure WS storage keys are consistent with RS storage keys.

If C-Store used conventional locking, then substantial lock contention would likely be observed, leading to very poor performance (read-only workloads)! They isolate read-only transactions using snapshot isolation. Snapshot isolation works by allowing read-only transactions to access the database as of some time in the recent past, before which we can guarantee that there are no uncommitted transactions. This avoids locks altogether.

Every tuple in WS is annotated with an insertion, and every tuple in RS and WS is annotated with a deletion time. A snapshot reading at time t can only read tuples inserted before t and deleted after t.

C-Store maintains a low and high watermark. Snapshot reads can be issued for any time

between the low and high watermark. If we let read-only transactions set their effective time arbitrarily, then we would have to support general time travel, an onerously expensive task.

They maintain an insertion vector (IV) for each projection segment in WS, which contains for each record the epoch in which the record was inserted. The tuple mover is programmed to ensure that no records in RS were inserted after the LWM, so RS doesn't have to maintain an insertion vector. Similarly, a deleted record vector is maintained (DRV).

Question 5.2. Describe the function of the insertion vector. Does it exist in RS and WS? Why?

The timestamps are coarse-grained epochs where each epoch lasts a couple of seconds. To increase epochs and the high water mark, a centralized timestamp authority decides to increment the epoch and does so after a round of communication ensuring all nodes have finished the last epoch. The latest completed epoch is the high water mark.

Read-write transactions use strict two-phase locking for concurrency control. Standard write-ahead logging is employed for recovery purposes; we use a NO-FORCE, STEAL policy but differ from the traditional implementation of logging and locking in that we only log UNDO records. We do not use strict two-phase commit, avoiding the PREPARE phase. Deadlock is detected via timeouts.

5.1.5 Recovery

C-Store maintains K-safety; i.e. sufficient projections and join indexes are maintained, so that K sites can fail within t, the time to recover, and the system will be able to maintain transactional consistency.

Question 5.3. Define K-safety.

A common case of failure is that the WS goes down but RS remains intact. This is common because RS is written to only by the tuple mover.

5.1.6 Tuple Mover

The job of the tuple mover is to move blocks of tuples in a WS segment to the corresponding RS segment, updating any join indexes in the process. It operates as a background task looking for worthy segment pairs. When it finds one, it performs a merge-out process, MOP on this (RS, WS) segment pair.

MOP will find all records in the chosen WS segment with an insertion time at or before the LWM, throwing out records deleted at or before LWM. MOP will create a new RS segment that we name RS'. Then, it reads in blocks from columns of the RS segment, deletes any RS items with a value in the DRV less than or equal to the LWM, and merges in column values from WS. The merged data is then written out to the new RS' segment, which grows as the merge progresses. After finishing the merge and updating join indexes, RS and RS' are swapped. The timestamp authority periodically increments the low watermark.

Question 5.4. Describe how the tuple mover works. What is the low watermark and high watermark?

5.1.7 Query Optimizer

Query plans in C-Store are built with the following operators: Decompress, Select, Mask, Project, Sort, Aggregation Operators, Concat, Permute (permutes a projection according to a join index), Join, Bitstring Operators.

The work-in-progress C-Store query optimizer differs from a traditional row-oriented query optimizer because it has to take into account (a) the cost of decompressing data vs operating on compressed data and (b) which projections to use to implement a query.

Question 5.5. How does the C-Store query optimizer differ from row-oriented query optimizers?

Question 5.6. Row-storage & B+ trees are write-optimized while other data structures like bit map indexes are read optimized. Why?

A row store will only store one copy of a relation and use a bunch of B+ tree indexes to access it. Because we only store a relation once, it can only be clustered on one set

of attributes. If we store multiple projects with different sort orders, we can traverse relations in different orders efficiently. Because the workload is read-only, maintaining these redundant copies is not expensive.

5.2 Hekaton: SQL Server's Memory-Optimized OLTP Engine [20]

Abstract

Hekaton is a new database engine optimized for memory resident data and OLTP workloads. Hekaton is fully integrated into SQL Server; it is not a separate system. To take advantage of Hekaton, a user simply declares a table memory optimized. Hekaton tables are fully transactional and durable and accessed using T-SQL in the same way as regular SQL Server tables. A query can reference both Hekaton tables and regular tables and a transaction can update data in both types of tables. T-SQL stored procedures that reference only Hekaton tables can be compiled into machine code for further performance improvements. The engine is designed for high concurrency. To achieve this it uses only latch-free data structures and a new optimistic, multiversion concurrency control technique. This paper gives an overview of the design of the Hekaton engine and reports some experimental results.

Hekaton is an embedded OLTP engine inside of Microsoft SQL Server, using compiled stored procedures, latch-free data structures, some optimistic MVCC, and more tricks for efficiently implementing transactions.

5.2.1 Design Considerations

An analysis done early on in the project drove home the fact that a 10-100X throughput improvement cannot be achieved by optimizing existing SQL Server mechanisms. Throughput can be increased in three ways: improving scalability, improving CPI (cycles per instruction), and reducing the number of instructions executed per request. The first 2 didn't do much. So they needed a way to drastically reduce the number of instructions executed per request.

- Optimize indexes for main memory
- Don't log index operations (just rebuild indexes entirely during recovery)
- Eliminate latches and locks
- Compile requests to native code
- Don't partition data (partitioning only works if the workload is also partitionable)

5.2.2 Storage and Indexing

Hash indexes and range indexes (via bw-trees, a lock-free version of B-trees) are supported. Each tuple contains 3 segments:

- Header with logical timestamps for begin and end for MVCC
- Links for indexes
- Tuple data

Every read operation specifies a logical (as-of) read time and only versions whose valid time overlaps the read time are visible to the read (via begin and end timestamps); all other versions are ignored. Different versions of a record always have non-overlapping valid times so at most one version of a record is visible to a read.

When a transaction deletes a tuple, it temporarily writes its transaction id into the tuples end timestamp, then later replaces it with its commit timestamp.

When a transaction inserts a tuple, it temporarily writes its transaction id in the begin timestamp. Once the transaction completes, it overwrites its id with its end timestamp. An update is like an a deletion followed by an insertion.

5.2.3 Programmability and Query Processing

Stored procedures are optimized into mixed abstract syntax trees (MAT) and then into pure imperative trees (PIT) and then into C code. Query plans are compiled to C code that doesn't use function calls; instead, it uses gotos and labels. The paper argues that this kind of code is smaller and faster. Complicated code for things like sorting is linked in and called via a function.

Compiled queries have some technical restrictions and limitations (e.g. if a table's schema is changed, stored procedures which operate on the table must be dropped). To overcome some of these restrictions, SQL Server allows regular/unrestricted/interpreted stored procedures to read and write Hekaton tables.

5.2.4 Transaction Management

Hekaton uses an optimistic multiversion concurrency control scheme for read uncommited, snapshot isolation, repeatable read, and serializable transactions without locking.

The simplest and most widely used MVCC method is snapshot isolation (SI). SI does not guarantee serializability because reads and writes logically occur at different times: reads at the beginning of the transaction and writes at the end. For repeatable read, we only need to guarantee read stability. For snapshot isolation or read committed, no validation at all is required. The paper discusses serializable transactions.

All records in the database contain two timestamps – begin and end. The begin timestamp denotes the commit time of the transaction that created the version and the end timestamp denotes the commit timestamp of the transaction that deleted the version (and perhaps replaced it with a new version). The valid time for a version of a record denotes the timestamp range where the version is visible to other transactions.

A transaction executing with logical read time RT must only see versions whose begin timestamp is less than RT and whose end timestamp is greater than RT.

Transactions first perform their reads and then perform their writes. The validation phase begins with the transaction obtaining an end timestamp. This end timestamp determines the position of the transaction within the transaction serialization history. Upon trying to commit, the transaction has to satisfy a couple of properties (validation):

- All the current read versions are the same as the ones read
- All scans can be repeated (no phantoms)

While validation may sound expensive, keep in mind that most likely the versions visited during validation remain in the L1 or L2 cache. Furthermore, validation overhead is lower for lower isolation: repeatable read requires only read validation and snapshot isolation and read committed require not validation at all.

Any transaction T1 that begins while a transaction T2 is in the validation phase becomes dependent on T2 if it attempts to read a version created by T2 or ignores a version deleted by T2. In that case T1 has two choices: block until T2 either commits or aborts, or proceed and take a commit dependency on T2. To preserve the nonblocking nature of Hekaton, we have T1 take a commit dependency on T2. This means that T1 is allowed to commit only if T2 commits. If T2 aborts, T1 must also abort so cascading aborts are possible.

Commit dependencies imply working with uncommitted data and such data should not be exposed to users. To solve this, there are read barriers—a transaction's result set is held back and not delivered to the client while the transaction has outstanding commit dependencies. The results are sent as soon as the dependencies have cleared.

Question 5.7. Describe commit dependencies in Hekaton with an example. Why are cascading aborts possible?

After a transaction commits, it updates the end timestamp of all the tuples it deleted and the begin timestamp of all the tuples it inserted. To do so efficiently, transactions maintain a write set pointing to these tuples.

5.2.5 Durability and Recovery

The data stored on external storage consists of transaction log streams and checkpoint streams.

- A log stream records all of the tuple versions inserted and deleted by an transaction during a period of logical time.
- Checkpoint streams come in two forms: a) data streams which contain all inserted versions during a timestamp interval, and b) delta streams, each of which is associated with a particular data stream and contains a dense list of integers identifying deleted versions for its corresponding data stream.

Each transaction produces a single record in the log stream once it commits. This is possible because Hekaton does not use write-ahead logging.

Question 5.8. Why doesn't Hekaton use WAL?

Question 5.9. Why isn't undo information logged? (Because it's an in-memory DB)

Log streams are stored in the regular SQL Server transaction log. Checkpoint streams are stored in SQL Server file streams which in essence are sequential files fully managed by SQL Server. Periodically, parts of the log are converted into data and delta streams and flushed to disk. Even more periodically, data and delta streams are merged together into more compressed streams that cover a larger period of logical time. Upon recovery, Hekaton processes data/delta stream pairs in parallel to collapse multiple data streams with very few remaining tuples.

5.2.6 Garbage Collection

First, a version becomes garbage if a) it was deleted (via explicit DELETE or through an UPDATE operation) by a committed transaction and b) the version cannot be read or

otherwise acted upon by any transaction in the system. A second, and less common way for versions to become garbage is if they were created by a transaction that subsequently rolls back.

A GC thread periodically scans the global transaction map to determine the begin timestamp of the oldest active transaction in the system. When the GC process is notified that it should begin collection, transactions committed or aborted since the last GC cycle are ordered by their end timestamps. Any transaction T in the system whose end timestamp is older than the oldest transaction watermark is ready for collection. More precisely, the versions deleted or updated by T can be garbage collected because they are invisible to all current and future transactions.

How to remove an object? In order for a garbage version to be removed it must first be unlinked from all indexes in which it participates. The GC subsystem collects these versions in two ways: (1) a cooperative mechanism used by threads running the transaction workload, and (2) a parallel, background collection process. Since regular index scanners may encounter garbage versions as they scan indexes, index operations are empowered to unlink garbage versions when they encounter them. If this unlinks a version from its last index, the scanner may also reclaim it.

This cooperative process naturally ensures that 'hot' regions of an index are constantly maintained to ensure they are free of obsolete versions. The background GC process cleans the "dusty corners."

Question 5.10. Describe the offline and online garbage collection processes in Hekaton.

5.3 Calvin: Fast Distributed Transactions for Partitioned Database Systems [52]

Abstract

Many distributed storage systems achieve high data access throughput via partitioning and replication, each system with its own advantages and tradeoffs. In order to achieve high scalability, however, today's systems generally reduce transactional support, disallowing single transactions from spanning multiple partitions. Calvin is a practical transaction scheduling and data replication layer that uses a deterministic ordering guarantee to significantly reduce the normally prohibitive contention costs associated with distributed transactions. Unlike previous deterministic database system prototypes, Calvin supports disk-based storage, scales near-linearly on a cluster of commodity machines, and has no single point of failure. By replicating transaction inputs rather than effects,

Calvin is also able to support multiple consistency levels—including Paxos-based strong consistency across geographically distant replicas—at no cost to transactional throughput.

Many distributed data stores (e.g., Dynamo, MongoDB) do not support transactions. In a traditional distributed transactional database, a transaction has to run some distributed commit protocol like 2PC which greatly increases the contention footprint (time locks are held). Calvin is a distributed database that supports ACID transactions without incurring 2PC or Paxos overheads.

5.3.1 Deterministic Database Systems

Calvin shrinks contention footprints by having nodes agree to commit a transaction before they acquire locks. Once they agree, they must execute the transaction as planned. They cannot abort.

Traditional commit protocols abort in the face of nondeterministic events (e.g., node failure) but don't have to. In order to avoid aborting a transaction in the face of node failure, Calvin runs the same transaction on multiple nodes. If any one of the nodes fail, the others are still alive to carry the transaction to fruition. When the failed node recovers, it can simply recover from another replica.

What is the problem here? If we execute the same batch of transactions on multiple nodes, they can execute in different orders. To prevent replicas from diverging, Calvin implements a deterministic concurrency control scheme which ensures that all replicas serialize all transactions in the same way. In short, Calvin predetermines a global order in which transactions should commit.

The other potential cause of an abort mentioned above—deterministic logic in the transaction (e.g. a transaction should be aborted if inventory is zero)—does not necessarily have to be performed as part of an agreement protocol at the end of a transaction. Rather, each node involved in a transaction waits for a one-way message from each node that could potentially deterministically abort the transaction, and only commits once it receives these messages.

Question 5.11. Explain deterministic and non-deterministic events—reasons for aborts.

5.3.2 System Architecture

Calvin is a piece of software you layer onto an existing storage system. There are 3 layers:

- The sequencing layer (or "sequencer") intercepts transactional inputs and places them into a global transactional input sequence—this sequence will be the order of transactions to which all replicas will ensure serial equivalence during their execution. The sequencer therefore also handles the replication and logging of this input sequence.
- The scheduling layer (or "scheduler") orchestrates transaction execution using a deterministic locking scheme to guarantee equivalence to the serial order specified by the sequencing layer while allowing transactions to be executed concurrently by a pool of transaction execution threads.
- The storage layer handles all physical data layout. Calvin transactions access data using a simple CRUD interface.

Question 5.12. What are the 3 layers in Calvin?

Sequencer and replication. Time is divided into 10 ms epochs. A sequencing node sequences and batches transaction requests into these epochs and then sends them to every scheduler in its replica. The scheduler round robin shuffles the sequences from every sequencing node in its replica.

Scheduling. Locking is done using 2PC with the minor variant requirement that locks are obtained in the pre-agreed upon global order. Transactions are written in C++, and read and write sets have to be set upfront. Schedulers broadcast all reads to all nodes participating in the transaction, and then all nodes execute the transaction. Some transactions don't know their read and write sets upfront, so they run optimistic lock location prediction (OLLP). It runs a query with loose consistency to guess the read and write sets and then submits it. If the read set changed, then the transaction is aborted.

5.3.3 Calvin with disk-based storage and Checkpointing

Even though Calvin executes transactions in a pre-agreed upon ordering, the disk fetches performed by transactions do not need to be run in this order. Calvin can optimistically prefetch data. Sequencers can also delay ordering a transaction until its resources will likely not already be locked.

Calvin can do a synchronous checkpoint in which an entire replica is frozen. Alternatively, Calvin can do a zig-zag checkpoint in which all writes after a point in the global log are written elsewhere. If the storage system supports multiversioning, then Calvin can perform a checkpoint using old versions of data.

[Shreya: TODO: read Paxos]

5.4 Spanner: Google's Globally-Distributed Database [15]

Abstract

Spanner is Google's scalable, multi-version, globally distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: nonblocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

Spanner supports full linearizable read-write transactions, read-only transaction, and snapshot read transactions.

5.4.1 Implementation

A (global) Spanner instance called a universe (e.g. test, dev, prod). Each universe contains multiple databases, and the data within a database is replicated across multiple zones. Each zone has: hundreds of span servers which store and serve data, a zone master that assigns data to span servers, and multiple location proxies that the span servers use to find the zone master.

Relations are divided into directories, and directories are divided into tablets. Tablets are replicated. Each tablet is replicated across a set of spanservers, and these spanservers form a Paxos group. One Paxos group is run per tablet, and the metadata and log are both stored in the tablet itself. The Paxos instance also uses 10-second leader leases. Writes to the tablet must go through Paxos. Reads can be read directly from a tablet so long as it is sufficiently up to date.

Unlike Bigtable, Spanner assigns timestamps to data, which is an important way in which Spanner is more like a multiversion database than a key-value store. A tablet's (100-1000 instances of data points) state is stored in a set of B-tree-like files and a write-ahead log, all on a distributed file system called Colossus (the successor to the Google File System. They use 2PL because transactions are usually long-lived.

5.4.2 Data Model

Spanner uses a hierarchical semi-relational data model. Relations must have a primary key. They can be nested within parents with child primary keys being prefixed by parent primary keys. These hierarchical relations are stored with rows interleaved and within the same directory.

5.4.3 Concurrency

Spanner offers four types of transactions: linearizable read-write transactions, snapshot (read-only) transactions, snapshot reads at a user specified timestamp, and snapshot reads with a user specified staleness bound.

Spanner guarantees that transactions are linearizable, so if a transaction T1 commits before transaction T2 starts, then the timestamp of T1 must be less than the timestamp of T2. To do so, it ensures that the timestamp of a transaction is between the actual start time and the actual end time.

The TrueTime API is implemented with a combination of GPS clocks and atomic clocks spread across multiple time masters within a data center. Periodically, time masters synchronize with one another. Time masters also check against their local clock, evicting themselves if there is too much drift. TrueTime clients poll multiple time masters and run Marzullo's algorithm.

5.4.4 Differences between Calvin and Spanner

- Spanner uses the TrueTime API for individual transactions to get bounds on time; Calvin uses the global sequencing layer to serialize transactions
- Spanner uses locking to get an ordering; Calvin's sequencer handles ordering
- Calvin replicates during sequencing; Spanner replicates after executing transaction
- 2PC cost is high in Spanner because each force-write involves a Paxos agreement; low in Calvin because transactions have deterministic execution and thus machine failures can simply replay the transaction instead of causing a huge abort

Performance differences:

- Spanner and Calvin have roughly equivalent latency for single-partition transactions, but Spanner has worse latency than Calvin for multi-partition transactions due to the extra phases in the transaction commit protocol.

- Both Calvin and Spanner can achieve very low snapshot-read latency
- Spanner has better latency than Calvin for read-only transactions submitted by clients that are physically close to the location of the leader servers for the partitions accessed by that transaction
- Calvin has higher throughput scalability than Spanner for transactional workloads where concurrent transactions access the same data. This advantage increases with the distance between datacenters.

Question 5.13. How does Spanner get consistency in the presence of transactions that touch data in multiple shards?

Spanner famously got around this downside with their TrueTime API. All transactions receive a timestamp which is based on the actual (wall-clock) current time. This enables there to be a concept of "before" and "after" for two different transactions, even those that are processed by completely disjoint set of servers. The transaction with a lower timestamp is "before" the transaction with a higher timestamp. Obviously, there may be a small amount of skew across the clocks of the different servers. Therefore, Spanner utilizes the concept of an "uncertainty" window which is based on the maximum possible time skew across the clocks on the servers in the system. After completing their writes, transactions wait until after this uncertainty window has passed before they allow any client to see the data that they wrote.

Question 5.14. Discuss some differences between Calvin and Spanner.

5.5 Building Efficient Query Engines in a High-Level Language [36]

Abstract

In this paper we advocate that it is time for a radical rethinking of database systems design. Developers should be able to leverage high-level programming languages without having to pay a price in efficiency. To realize our vision of abstraction without regret, we present LegoBase, a query engine written in the high-level programming language Scala. The key technique to regain efficiency is to apply generative programming: the Scala code that constitutes the query engine, despite its high-level appearance, is actually a program generator that emits specialized, low-level C code. We show how the combination of

high-level and generative programming allows to easily implement a wide spectrum of optimizations that are difficult to achieve with existing low-level query compilers, and how it can continuously optimize the query engine. We evaluate our approach with the TPC-H benchmark and show that: (a) with all optimizations enabled, our architecture significantly outperforms a commercial in-memory database system as well as an existing query compiler, (b) these performance improvements require programming just a few hundred lines of high-level code instead of complicated low-level code that is required by existing query compilers and, finally, that (c) the compilation overhead is low compared to the overall execution time, thus making our approach usable in practice for efficiently compiling query engines.

Note 5.1. This paper was really hard to understand out-of-context. Read another paper by the same author(s), "How to Architect a Query Compiler," first [48].

Despite the differences between the individual approaches, all compilation frameworks generate an optimized query evaluation engine on-the-fly for each incoming SQL query. There are 4 problems with this:

- Template expansion misses optimization potential
- Template expansion is brittle and hard to implement (developers must deal with low level codegen)
- Limited scope of query compilation (only optimizes queries, not other DB code)
- Limited adaptivity

LegoBase compiles a query plan (also written in Scala) into a C program that can execute the query plan. LegoBase performs a number of optimizations that other query compilers do not do. Plus, it's written in a high-level language!

5.5.1 Example

```
SELECT *
FROM R, (SELECT S.D,
              SUM(1-S.B) AS E,
              SUM(S.A*(1-S.B)),
              SUM(S.A*(1-S.B)*(1+S.C))
```

```
      FROM S
      GROUP BY S.D) T
WHERE R.Z = T.E AND R.B = 3
```

A traditional query optimizer will miss some things: (1) can eliminate common subexpresions (like 1-S.B)), (2) can reuse hash tables on the same relation for different operators.

5.5.2 General Execution Workflow

To obtain a physical plan, we pass the incoming query through any existing query optimizer Then, we pass the generated physical plan to LegoBase. It then takes the query plan and replaces each operator with a LegoBase operator. It then compiles the heck out of the query plan using LMS and other tricks to produce a C program which can execute the query.

This architecture also allows LegoBase to re-compile queries based on changing runtime information. For example, the compiled code does not need to have any conditionals checking whether a log flag is set. Instead, LegoBase can recompile the code every time the log flag is toggled.

5.5.3 Optimizations

Pull to push: Some have argued that a Volcano-style pull based model of execution incurs unnecessary overhead. A faster approach is for data to be repeatedly pushed upwards through a tree until it is materialized (e.g. in a hash table somewhere). LegoBase converts pull based operators into push based operators by making the child's next also invoke the parent's next. [Shreya: but isn't the exchange operator push based?]

Eliminating redundant materializations: pattern match on a query plan and eliminate redundant materializations. For example, one pattern might match hash joins whose left child is an aggregate and replace it with a single operator that only materializes the aggregates once

Data structure specialization: convert hash tables to arrays whose size is set based on runtime estimates of query size

Changing data layout: automatically convert operators that use a row-oriented data layout into operators that use a column-oriented data layout

Question 5.15. Give an example of a query that builds redundant hashtables.

Answer: a subquery that performs an aggregation on a table using a hash table, and then joining the result of that subquery on another table (recreating the hashtable)

6 Distributed Data, Weak Isolation, Relaxed Consistency

The following papers are covered:

- Transaction Management in the R* Distributed Database Management System
- Generalized Isolation Level Definitions
- Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System
- Dynamo: Amazon's Highly Available Key-value Store
- CAP Twelve Years Later: How the "Rules" Have Changed
- Consistency Analysis in Bloom: a CALM and Collected Approach

6.1 Transaction Management in the R* Distributed Database Management System [40]

Abstract

This paper deals with the transaction management aspects of the R* distributed database system. It concentrates primarily on the description of the R* commit protocols, Presumed Abort (PA) and Presumed Commit (PC). PA and PC are extensions of the well-known, two-phase (2P) commit protocol. PA is optimized for read-only transactions and a class of multisite update transactions, and PC is optimized for other classes of multisite update transactions. The optimizations result in reduced intersite message traffic and log writes, and, consequently, a better response time. The paper also discusses R*'s approach toward distributed deadlock detection and resolution.

This paper focuses on the performance aspects of distributed commit protocols. They suggest that complicated protocols developed for dealing with rare kinds of failures during commit coordination are not worth the costs that they impose on the processing of distributed transactions during normal times (i.e., when no failures occur). Consequently, they extended the conventional 2P commit protocol to support a tree of processes and defined the Presumed Abort (PA) and the Presumed Commit (PC) protocols to improve the performance of distributed transaction commit.

6.1.1 Properties of an Ideal Atomic Commit Protocol

1. Guaranteed transaction atomicity always
2. Ability to "forget" outcome of commit processing after a short amount of time
3. Minimal overhead in terms of log writes and message traffic
4. Optimized no-failure performance
5. Exploitation of completely or partially read-only transactions
6. Maximizing the ability to perform unilateral aborts

6.1.2 The Two-Phase Commit Protocol

There is one process, called the coordinator, that is connected to the user application and a set of other processes, called the subordinates. During the execution of the commit protocol the subordinates communicate only with the coordinator, not among themselves.

2PC Normally Without Considering Failures.

In the first round, the coordinator sends prepare messages to all subordinates. Every subordinate returns either a vote yes to commit (after force-writing a prepare) or a vote no to abort. In the second round, the coordinator force-writes a commit and sends a commit message to all subordinates if they all voted yes. Otherwise, if any subordinate voted no, the coordinator sends an abort message to all the subordinates that voted yes. Then, subordinates force-write the commit or abort and then send back ACKs to the coordinator. The coordinator will then write an end in its own time.

Considering Failures.

We assume that at each active site a recovery process exists and that it processes all messages from recovery processes at other sites and handles all the transactions that were executing the commit protocol at the time of the last failure of the site.

Consider a subordinate crash. If it did not get to the prepare state, it will abort the transaction. If it got to the prepare state but not the commit state, it should ask the coordinator how to proceed. If it got to the commit/abort state, it should commit or abort.

Consider a coordinator crash. If it did not send out a prepare, it aborts the transaction. If it sent out a prepare but did not send out a commit/abort, it can either resend the prepare or abort. If it sent out a commit/abort (but no end), it either proceeds with the commit or aborts.

Question 6.1. Why must the subordinate force-write commit/prepare before sending a message back to the coordinator?

6.1.3 Hierarchical 2PC

Suppose we have trees of processes. Each process communicates directly with only its immediate neighbors in the tree, that is, parent and children. In fact, a process would not even know about the existence of its nonneighbor processes. The root of the tree is the coordinator, the leaves of the tree are subordinates, and the internal nodes play the role of both.

The inner nodes forward prepare messages down the tree and aggregate votes up the tree. That is, if all of an inner node's children vote yes, the node forwards a yes upward. If any of the children vote no, the inner node forwards a no upward and also sends abort messages downward. When an inner node receives a commit or abort message, it force writes it, sends an acknowledgement upward, and forwards the abort or commit downward.

Question 6.2. What order of operations does an intermediate node perform in hierarchical 2PC when it receives a commit or abort message?

The Presumed Abort Protocol. Presumed abort recognizes that if a coordinator doesn't know about a transaction, it presumes that it aborted. This means that the abort record need not be forced (both by the coordinator and each of the subordinates), and no ACKs need to be sent (by the subordinates) for ABORTS. This allows us to avoid sending some messages and avoid force writing some log entries.

Let us now consider completely or partially read-only transactions and see how we can take advantage of them.

Definition 6.1. A transaction is **partially read-only** if some processes of the transaction do not perform any updates to the database while the others do.

If a leaf process receives a PREPARE and it finds that it has not done any updates (i.e., no UNDO/REDO log records have been written), then it sends a READ VOTE, releases its

locks, and "forgets" the transaction. The subordinate writes no log records. Why? As far as it is concerned, it does not matter whether the transaction ultimately gets aborted or committed. So coordinators don't need to send it a COMMIT or ABORT. If a coordinator receives all read votes, it also forgets about the transaction and logs nothing.

To summarize, for a (completely) read-only transaction, none of the processes write any log records, but each one of the nonleaf processes sends one message (PREPARE) to each subordinate and each one of the nonroot processes sends one message (READ VOTE). For a committing partially read-only transaction, the root process sends two messages (PREPARE and COMMIT) to each update subordinate and one message (PREPARE) to each of the read-only subordinates. Each one of the nonleaf, nonroot processes that is the root of an update subtree sends two messages (PREPARE and COMMIT) to each update subordinate, one message (PREPARE) to each of the other subordinates, and two messages (YES VOTE and ACK) to its coordinator. Each one of the nonleaf, nonroot processes that is the root of a read-only subtree behaves just like the corresponding processes in a completely read-only transaction following presumed abort (PA).

Question 6.3. Explained presumed aborts and how this can lead to optimizations in read-only transactions (both completely and partially RO).

Question 6.4. How many messages does a coordinator have to send? Intermediate nodes? Leaf nodes? In a partially read-only transaction?

6.1.4 The Presumed Commit Protocol

Since most transactions are expected to commit, it is only natural to wonder if, by requiring ACKs for ABORTS, commits could be made cheaper by eliminating the ACKs for COMMITS. Can we make the no-information case the COMMIT scenario? Here, if a coordinator doesn't know about a transaction, then it assumes that it is a commit.

One extension has to be made. To see this, consider the case where a coordinator sends a PREPARE, then crashes. When it recovers, it will have forgotten about the transaction and moved on, since no records were logged. However, the subordinates will have received silence and simply committed on their own time, causing inconsistency.

To fix this, the coordinator has to force-write a PREPARE with the subordinate IDs before sending the PREPARE message. Then if the coordinator crashes after sending a PREPARE, it can recover and send aborts to the relevant parties and receive the ACK.

A nonleaf process behaves in PC similar to PA except:

1. At the start of the first phase (i.e., before sending the PREPARES) it force-writes a collecting record, which contains the names of all the subordinates, and moves into the collecting state
2. It force-writes only abort records (except in the case of the root process, which force-writes commit records also)
3. It requires ACKs only for ABORTS and not for COMMITS;
4. It writes an end record only after an abort record (if the abort is done after a collecting record is written) and not after a commit record
5. Only when in the aborting state will it, on noticing a subordinate's failure, hand over the transaction to the restart process
6. In the case of a (completely) read-only transaction, it would not write any records at the end of the first phase in PA, but in PC it would write a commit record and then "forget" the transaction

Question 6.5. What does it mean to forget about a transaction?

Question 6.6. Discuss how the presumed commit protocol works, and a modification we must make to ensure consistency.

Question 6.7. Discuss different types of workloads in which PA and PC would outperform the other.

PA strictly better than 2P; PA better than PC for completely read-only (saving the coordinator two log writes, including a force), and partially-read-only if only coordinator does update (saving the coordinator a force-write). If only one update subordinate, PA and PC equal in terms of log writes, but PA needs extra ACK. For n¿1 update subordinates, PC and PA need same number of log records, but PA will force n-1 times while PC will not (because of commit records by subordinates). PA will also send n extra messages (due to ACKs).

6.1.5 Deadlock

In R*, there is no separate lock manager process—if a node detects a deadlock (after getting some waits-for information from another node), it aborts a local transaction (the overhead of trying to find the most optimal transaction across all the graphs is too much, because then you need to coordinate the distributed abort). In R*, there is one deadlock detector (DD) at each site. The DDs at different sites operate asynchronously. The frequencies at which local and global deadlock detection searches are initiated can vary from site to site. Each DD wakes up periodically and looks for deadlocks after gathering the wait-for information from the local DBMS and the communication manager.

Depending on whether or not (1) the wait-for information transmission among different sites is synchronized and (2) the nodes of the wait-for graph are transactions or individual processes of a transaction, false deadlocks might be detected. In R* transmissions are not synchronized and the nodes of the graph are transactions. Since we do not expect false deadlocks to occur frequently, we treat every detected deadlock as a true deadlock.

6.2 Generalized Isolation Level Definitions [1]

Abstract

Commercial databases support different isolation levels to allow programmers to trade off consistency for a potential gain in performance. The isolation levels are defined in the current ANSI standard, but the definitions are ambiguous and revised definitions proposed to correct the problem are too constrained since they allow only pessimistic (locking) implementations. This paper presents new specifications for the ANSI levels. Our specifications are portable; they apply not only to locking implementations, but also to optimistic and multi-version concurrency control schemes. Furthermore, unlike earlier definitions, our new specifications handle predicates in a correct and flexible manner at all levels.

This paper gives new, precise definitions of the ANSI-SQL isolation levels. Previous definitions were ambiguous and not implementation-independent (e.g., specific to locking.) This paper introduces implementation-independent unambiguous isolation level definitions.

6.2.1 Previous Work

Gray's degrees of consistency:

- Degree 0: short write locks, doesn't overwrite dirty data from other transactions

- Degree 1: short write locks & short read locks, read uncommitted
- Degree 2: long write locks & short read locks, read committed
- Degree 3: long write locks & long read locks: repeatable read & serializable

ANSI-SQL standard: ambiguous and wrong. These definitions only allow histories that would occur in a system using long/short read/write item/predicate locks.

- P0: dirty write, w1(x) and w2(x)
- P1: dirty read, w1(x) and r2(x)
- P2: non-repeatable read, r1(x) and w2(x)
- P3: phantom, r1(P) and w2(y in P)

Question 6.8. Discuss Gray's consistency levels and ANSI-SQL levels. How do they map to each other? How do they map to isolation levels READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE?

6.2.2 Restrictiveness of Preventative Approach

As mentioned, this approach disallows all histories that would not occur in a locking scheme and prevents conflicting operations from executing concurrently.

Phenomenon P0 can occur in optimistic implementations since there can be many uncommitted transactions modifying local copies of the same object concurrently; if necessary, some of them will be forced to abort so that serializability can be provided. Thus, disallowing P0 can rule out optimistic implementations.

Condition P1 precludes transactions from reading updates by uncommitted transactions. Such reads are disallowed by many optimistic schemes, but they are desirable in mobile environments, where commits may take a long time if clients are disconnected from the servers.

Proscribing phenomenon P2 disallows a modification to an object that has been read by an uncommitted transaction (P3 rules out a similar situation with respect to predicates). There is no harm in allowing phenomenon P2 if transactions commit in the right order.

6.2.3 Database Model and Transaction Histories

Every object in the DB has one or more versions, but transactions interact with objects. When a transaction writes an object x, it creates a new version of x. A transaction Ti can modify an object multiple times; its first modification of object x is denoted by xi:1, the second by xi:2, and so on. Version xi denotes the final modification of x performed by T i before it commits or aborts. A transaction's last operation, commit or abort, indicates whether its execution was successful or not; there is at most one commit or abort operation for each transaction.

The committed state reflects the modifications of committed transactions. When transaction Ti commits, each version xi created by Ti becomes a part of the committed state and we say that Ti installs xi ; the system determines the ordering of xi relative to other committed versions of x. If Ti aborts, xi does not become part of the committed state.

A history H over a set of transactions consists of two parts — a partial order of events E that reflects the operations (e.g., read, write, abort, commit) of those transactions, and a version order of objects, $<<$, that is a total order on committed versions of each object.

Partial orders obey these rules:

- It preserves the order of all events within a transaction including the commit and abort events
- $r_j(x_{i.m})$ in E is always preceded by $w_i(x_{i.m})$
- if there is an $w_i(x_{i.m})$ in E, any successive $r_i(x_j)$ will be $x_j = x_{i.m}$ if there is no intervening $w_i(x_{i.n})$ (it reads its latest writes)
- History is complete: all transactions' events in E have either a commit or abort event. A history that is not complete can be completed by appending abort events for uncommitted transactions in E

The second part of a history H is the version order $<<$, which obeys these rules:

- the version order of each object x contains exactly one x_{init} and at most one committed dead version x_{dead} (deleted version)
- x_{init} is the first version in the version order, and if x_{dead} exists it is the last version in the version order
- if $r_j(x_i)$ exists in a history, then x_i is a visible version

Conflicts Name	*Description* (T_j conflicts on T_i)	Notation in DSG
Directly write-depends	T_i installs x_i and T_j installs x's next version	$T_i \xrightarrow{ww} T_j$
Directly read-depends	T_i installs x_i, T_j reads x_i or T_j performs a predicate-based read, x_i changes the matches of T_j's read, and x_i is the same or an earlier version of x in T_j's read	$T_i \xrightarrow{wr} T_j$
Directly anti-depends	T_i reads x_h and T_j installs x's next version or T_i performs a predicate-based read and T_j overwrites this read	$T_i \dashrightarrow T_j$

Figure 3: Definitions of direct conflicts between transactions (Adya et al.)

Question 6.9. In a history, what is the difference between the partial order and the version order?

Predicate reads return a set of versions for each object being read. Reading and writing these versions happens in subsequent, separate read and write operations.

6.2.4 Conflicts and Serialization Graphs

We define three kinds of direct conflicts that capture conflicts of two different committed transactions on the same object or intersecting predicates.

- Read dependencies (wr): either direct item-read-depends or predicate-read-depends
- Anti dependencies (rw): either direct item-anti-depends or predicate-anti-depends
- Write dependencies (ww): there is no notion of predicate-write-depends since predicate-based modifications are modeled as queries followed by writes on individual tuples

Isolation levels are defined as constraints on serialization graphs, constructed by rules in Figure 3. . Each node in the graph corresponds to a committed transaction and directed edges correspond to different types of direct conflicts.

Dependency graphs don't include aborted transactions.

Question 6.10. What is the difference between read dependencies and anti dependencies?

Level	Phenomena disallowed	Informal Description (T_i can commit only if:)
PL-1	G0	T_i's writes are completely isolated from the writes of other transactions
PL-2	G1	T_i has only read the updates of transactions that have committed by the time T_i commits (along with PL-1 guarantees)
PL-2.99	G1, G2-item	T_i is completely isolated from other transactions with respect to data items and has PL-2 guarantees for predicate-based reads
PL-3	G1, G2	T_i is completely isolated from other transactions, i.e., all operations of T_i are before or after all operations of any other transaction

Figure 4: Summary of portable ANSI isolation levels (Adya et al.)

6.2.5 New Generalized Isolation Specifications

They define each isolation level in terms of phenomena that must be avoided at each level. Phenomena are prefixed by "G" to denote the fact that they are general enough to allow locking and optimistic implementations. They refer to the new levels as PL levels (where PL stands for "portable level").

- G0: No cycles consisting of only ww edges.
- G1: No reading data from aborted transactions/trigger cascading aborts (G1a), no reading the intermediate values of transactions (G1b), and no cycles consisting of only wr and ww edges (G1c).
- G2: No cycles consisting of only wr, ww, and rw cycles
- G2-item: No cycles consisting of only wr, ww, and item-rw cycles (can have predicate rw cycles)

- PL-1: G0 (read uncommitted)
- PL-2: G1 (read committed)
- PL-2.99: G1, G2-item (repeatable read)
- PL-3: G1, G2 (serializable)

See Figure 4 for a description of isolation levels.

6.2.6 Mixing Isolation Levels

In a mixed system, each transaction specifies its level when it starts and this information is maintained as part of the history and used to construct a mixed serialization graph or MSG. MSG contains nodes corresponding to committed transactions and edges corresponding to dependencies, but only dependencies relevant to a transaction's level or obligatory dependencies show up as edges in the graph.

Definition 6.2. A history H is **mixing-correct** if MSG(H) is acyclic and phenomena G1a and G1b do not occur for PL-2 and PL-3 transactions. If a history is mixing-correct, each transaction is provided the guarantees that pertain to its level.

6.3 Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System [51]

Abstract

Bayou is a replicated, weakly consistent storage system designed for a mobile computing environment that includes portable machines with less than ideal network connectivity. To maximize availability, users can read and write any accessible replica. Bayou's design has focused on supporting application-specific mechanisms to detect and resolve the update conflicts that naturally arise in such a system, ensuring that replicas move towards eventual consistency, and defining a protocol by which the resolution of update conflicts stabilizes, It includes novel methods for conflict detection, called dependency checks, and per-write conflict resolution based on client-provided merge procedures. To guarantee eventual consistency, Bayou servers must be able to rollback the effects of previously executed writes and redo them according to a global serialization order. Furthermore, Bayou permits clients to observe the results of all writes received by a server, including tentative writes whose conflicts have not been ultimately resolved. This paper presents the motivation for and design of these mechanisms and describes the experiences gained with an initial implementation of the system.

Users working concurrently can introduce conflicts, especially with unreliable network connections. Can we balance realistic expectations of poor network connectivity with the desire to maintain weakly consistent, replicated data? Unlike previous systems, Bayou uses application-specific logic (dependency checks, merge procedures) to detect and resolve conflicts of concurrent writes.

Bayou has 2 main contributions: (1) its application specific logic to let users define what a conflict is, and (2) eventual consistency through tentative and committed states of an

update. There are 2 motivating applications: calendar reservation system for meeting rooms and a Google Scholar-type bibliography management system.

6.3.1 System Model

Each data collection is replicated in full at a number of servers. Applications running as clients interact with the servers through the Bayou application programming interface (API), which is implemented as a client stub bound with the application. API supports read and write operations. Access to one server is enough for a client to perform read-/write operations.

Write operations contain information on how to deal with conflicts. Each write has a globally unique writeID. Storage system maintains a totally ordered log of writes, which are passed from one server to another via pairwise contacts (called ”anti-entropy sessions”). Writes will make it to all the servers eventually unless servers are permanently partitioned.

6.3.2 Conflict Detection and Resolution

System implements the mechanisms for reliably detecting conflicts, as specified by the application, and for automatically resolving them when possible. The Bayou system includes two mechanisms for automatic conflict detection and resolution that are intended to support arbitrary applications: dependency checks and merge procedures. Every write is bundled with a dependency check and a merge procedure.

A dependency check is a SQL query over the DB and an accompanying expected result (e.g., make sure no other user has reserved the room for the requested time). Bayou executes the dependency check. If the result is expected, then it applies the write. Otherwise, the user-defined merge procedure is run (e.g., book alternative meeting times).

Question 6.11. What are dependency checks and merge procedures? Why are they useful? Give an example of each.

6.3.3 Replica Consistency

While the replicas held by two servers at any time may vary in their contents because they have received and processed different Writes, a fundamental property of the Bayou

design is that all servers move towards eventual consistency. But there are no bounds enforced on write propagation delays.

Two important features of the Bayou system design allows servers to achieve eventual consistency. First, Writes are performed in the same, well-defined order at all servers. Second, the conflict detection and merge procedures are deterministic so that servers resolve the same conflicts in the same manner. In practice, because Bayou's Write operations include arbitrary merge procedures, it is effectively impossible either to determine whether two Writes commute.

Question 6.12. What properties of Bayou allow it to reach eventual consistency? Why do we need a total ordering of operations (e.g., why can't we leverage commutativity of writes)?

When a Write is accepted by a Bayou server from a client, it is initially deemed tentative. Tentative Writes are ordered according to timestamps assigned to them by their accepting servers. The only requirement placed on timestamps for tentative Writes is that they be monotonically increasing at each server.

Because servers may receive Writes from clients and from other servers in an order that differs from the required execution order, and because servers immediately apply all known Writes to their replicas, servers must be able to undo the effects of some previous tentative execution of a Write operation and reapply it in a different order.

Conceptually, each server maintains a log of all Write operations that it has received, sorted by their committed or tentative timestamps, with committed Writes at the head of the log.

6.3.4 Write Stability and Commitment

A Write is said to be stable at a server when it has been executed for the last time by that server. Clients can query servers to see which writes have been committed.

One server, a primary server, has the authority to commit updates. This is preferred rather than a majority model, since nodes might frequently be offline. All writes are globally ordered, stable writes before tentative writes.

6.3.5 Storage System Implementation Issues

There is a write log, tuple store, and undo log. Every server maintains a sequence of all writes ordered by id, committed writes appearing before tentative writes. As an optimization, the servers can throw away a prefix of the committed writes because they will never again be executed. If they use this optimization, they must also record the last time that they discarded writes so they don't reintroduce the writes via anti-entropy sessions (timestamps stored via an O-vector on each server, of size num servers).

The tuple store is an in-memory relational database which supports a subset of SQL. The database must present a view of the committed database as well as the tentative database. To do so, every tuple is annotated with a committed and tentative bit. Queries propagate these bits to outputs.

The undo log is not well-described.

Question 6.13. What are the 3 data structures stored in Bayou?

6.3.6 Access Control

Can't rely on central auth server because of connectivity issues. So public key cryptography is used to handle security issues on each server.

6.4 Dynamo: Amazon's Highly Available Key-value Store [17]

Abstract

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems. This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of

object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Amazon needs availability over consistency, so they built Dynamo.

6.4.1 System Interface

Dynamo is a key-value store where the values are arbitrary blobs of data. Users can issue get and put requests. Get requests return either an object or list of conflicting objects and context. User has to merge the multiple objects. Put requests also have a context parameter, where context can maintain version clocks.

6.4.2 Partitioning Algorithm

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes. It relies on consistent hashing: the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

Each node is assigned some number of virtual nodes (i.e. spots in a circular 128-bit key space) and is assigned all data between it and its predecessor going counterclockwise. The number of virtual nodes that a physical node is assigned can be varies based on the capacity of the physical node (for load balancing).

6.4.3 Replication

When data is sent to a coordinator, it writes the data locally and replicates the data to N clockwise (non-virtual) neighbors in the ring. The list of nodes that is responsible for storing a particular key is called the preference list. Every node in the system can determine which nodes should be in this list for any particular key. To account for node failures, preference list contains more than N nodes.

Question 6.14. What is a preference list?

6.4.4 Data Versioning

Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation). However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the client must perform the merge conflict (collapse multiple branches of data evolution).

Dynamo versions key-value pairs with vector clocks. A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object. Users can analyze the vector clocks for versions of objects to perform reconciliation. If vector clocks get too long, old entries are thrown out.

If a write a happens before a write b, then the two writes can be reconciled trivially; this is known as syntactic reconciliation. However, if a and b are concurrent, then the system or the user has to perform semantic reconciliation.

Question 6.15. What is the difference between semantic and syntactic reconciliation?

6.4.5 Execution of get () and put () operations

A client can either issue a request to a load balancer or issue it itself if it is partition aware. A client issues a get or put to a load balancer then selects a node in the ring. If the node is in the preference list for the key, then it services the request. Otherwise, the request is forwarded to the first node in the preference list.

When a node in the preference list receives a get request, it forwards the request to R replicas in the preference list, including itself, collects all the versions and returns them to the user. When a node in the preference list receives a put request, it performs the write locally and then forwards the write to W-1 other replicas in the preference list.

Dynamo uses quorums to write data. A read must be acknowledged by R servers, a write must be acknowledged by W servers, and $R + W > N$.

6.4.6 Handling Failures

Hinted handoff: Remember that the preference list contains more than N entries even though the data is only replicated on N machines. If we go to do a write and less than

W of the replicas are available, then we look for other non-replicas in the preference list. These nodes will eventually relay the writes back to the replicas. This technique is known as a sloppy quorum.

Question 6.16. What is the sloppy quorum technique and why is it useful?

Hinted handoff works best if the system membership churn is low and node failures are transient. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized.

To detect permanent failures, nodes store merkle trees of their data.

6.4.7 Membership and Failure Detection

In Amazon's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas.

To add a new node into the ring, an admin manually contacts an existing Dynamo node. All nodes maintain a history of nodes entering and leaving the system and information about the key ring assignments.

When a node starts for the first time, it chooses its set of tokens (virtual nodes in the consistent hash space) and maps nodes to their respective token sets. Mapping is persisted on disk.

A node conservatively guesses that another node has failed if it doesn't respond for a while. There is no global failure detection. Seed nodes periodically communicate with each other to ensure the ring is not split.

6.4.8 Experiences and Lessons Learned

- Support business logic specific reconciliation
- Improving durability decreases performance
- Ensuring uniform load is hard
- In practice, there aren't that many divergent data items

- Instead of communicating with a load balancer, clients can periodically request membership information and route requests themselves
- Dynamo implements admission control for background tasks so they don't interfere with foreground tasks

Question 6.17. Explain the virtual node setup in Dynamo.

6.5 CAP Twelve Years Later: How the "Rules" Have Changed [9]

Abstract

The CAP theorem asserts that any networked shared-data system can have only two of three desirable properties. However, by explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some tradeoff of all three.

The CAP theorem dictates that in the face of network partitions, replicated data stores must choose between high availability and strong consistency. This is a 12 year retrospective saying it is misleading now for a few reasons:

- Because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned
- The choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved
- All three properties are more continuous than binary (various levels of consistency)

6.5.1 CAP-Latency Connection

The partition decision (during a timeout, when the program must make a fundamental decision) is where the CAP theorem really takes place. Either you cancel the operation and decrease availability, or you proceed with the operation and risk inconsistency.

Partitions are time bounds on communication. This pragmatic view gives rise to several important consequences. The first is that there is no global notion of a partition, since some nodes might detect a partition, and others might not. The second consequence is

that nodes can detect a partition and enter a partition mode—a central part of optimizing C and A. Finally, this view means that designers can set time bounds intentionally according to target response times; systems with tighter bounds will likely enter partition mode more often and at times when the network is merely slow and not actually partitioned.

6.5.2 Managing Partitions

There are 3 steps to handle partitions:

- Detect the start of a partition
- Enter explicit partition mode that can limit some operations
- Initiate partition recovery when communication is restored

6.5.3 Which operations should proceed?

More generally, partition mode gives rise to a fundamental user-interface challenge, which is to communicate that tasks are in progress but not complete. Researchers have explored this problem in some detail for disconnected operation, which is just a long partition.

The best way to track the history of operations on both sides is to use version vectors, which capture the causal dependencies among operations. The vector's elements are a pair (node, logical time), with one entry for every node that has updated the object and the time of its last update. Dynamo does this!

6.5.4 Partition Recovery

Once a system recovers from a partition it has to make the state consistent again and compensate for any mistakes made during the partition. Sometimes a system is unable to automatically make the state consistent and depends on manual intervention. Other systems can automatically restore consistency (for example, through use of CRDTs).

6.6 Consistency Analysis in Bloom: a CALM and Collected Approach [4]

Abstract:

Distributed programming has become a topic of widespread interest, and many programmers now wrestle with tradeoffs between data consistency, availability and latency. Distributed transactions are often rejected as an undesirable tradeoff today, but in the absence of transactions there are few concrete principles or tools to help programmers design and verify the correctness of their applications. We address this situation with the CALM principle, which connects the idea of distributed consistency to program tests for logical monotonicity. We then introduce Bloom, a distributed programming language that is amenable to high-level consistency analysis and encourages order-insensitive programming. We present a prototype implementation of Bloom as a domain-specific language in Ruby. We also propose a program analysis technique that identifies points of order in Bloom programs: code locations where programmers may need to inject coordination logic to ensure consistency. We illustrate these ideas with two case studies: a simple key-value store and a distributed shopping cart service.

This paper introduces the CALM conjecture and Bloom, a disorderly declarative programming language based on CALM, which allows users to write loosely consistent systems in a more principled way.

How would we know our program is eventually consistent? The CALM conjecture embraces Consistency As Logical Monotonicity and says that logically monotonic programs are eventually consistent for any ordering and interleaving of message delivery and computation. Moreover, they do not require waiting or coordination to stream results to clients.

Monotonic programs—e.g., programs expressible via selection, projection and join (even with recursion)—can be implemented by streaming algorithms that incrementally produce output elements as they receive input elements. The final order or contents of the input will never cause any earlier output to be "revoked" once it has been generated. Non-monotonic programs—e.g., those that contain aggregation or negation operations—can only be implemented correctly via blocking algorithms that do not produce any output until they have received all tuples in logical partitions of an input set.

Monotonic programs guarantee eventual consistency under any interleaving of delivery and computation. By contrast, non-monotonicity—the property that adding an element to an input set may revoke a previously valid element of an output set—requires coordination schemes that "wait" until inputs can be guaranteed to be complete.

6.6.1 BUD: Bloom Under Development

Bud (Bloom under development) is the realization of the CALM theorem. It is Bloom implemented as a Ruby DSL. Users declare a set of Bloom collections (e.g. persistent tables, temporary tables, channels) and an order independent set of declarative statements

similar to Datalog.

Bloom execution proceeds in timesteps, and each timestep is divided into three phases. First, external messages are delivered to a node. Second, the Bloom statements are evaluated. Third, messages are sent off elsewhere. Bloom also supports modules and interfaces to improve modularity.

[Shreya: TODO: read more]

7 Parallel Dataflow

Papers covered:

- Parallel Database Systems: The Future of High Performance Database Processing
- Encapsulation of Parallelism in the Volcano Query Processing System
- MapReduce: simplified data processing on large clusters
- TAG: A tiny aggregation service for ad-hoc sensor networks
- Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing

7.1 Parallel Database Systems: The Future of High Performance Database Processing [18]

In the past, people thought specialized hardware was going to overtake parallel databases. Well, highly parallel database systems are winning (as of 1992). Their hypothesis was that relational queries are highly suited to parallel execution (uniform operations applied to uniform sequences of data).

There are 2 definitions of parallelism: Pipelined parallelism occurs when multiple operators that pipe into one another are implemented in parallel. Partitioned Parallelism occurs when portions of the same SQL query can be executed in parallel on different data on different machines.

Question 7.1. What's the difference between pipelined and partitioned parallelism?

Parallel systems are evaluated in two ways: linear speedup (2x hardware can perform the same workload in half elapsed time), and linear scaleup (2x workload can be performed with 2x hardware in the same amount of time). There are 2 kinds of scaleup, batch and transactional. The generic barriers to scaleup and speedup are the "triple threats" of:

- Startup: there is a startup overhead to starting a parallel operation (e.g., thousands of processes need to spawn)
- Interference: the slowdown each new process imposes on others when accessing shared resources

- Skew: As the number of parallel steps increases, the variance of sizes of steps increases. The job time is greater than or equal to the time of the slowest step.

Question 7.2. What are the 3 barriers to scaleup and speedup?

7.1.1 Trend to Shared Nothing DBs

There are 3 ways to achieve parallel memory: shared-memory, shared-disk, and shared-nothing. In a shared-memory system, multiple threads or processes share a common memory and a common set of disks. In a shared-disk, multiple processes don't share memory, but they share disks. In shared-nothing, nodes don't share anything. Stonebraker says the third is the one that really scales.

Old software is the most significant barrier to parallelism.

7.1.2 A Parallel Dataflow Approach to SQL Software

Select-project operators are also known as scans. The benefits of pipeline parallelism are limited because of three factors:

1. Relation pipelines are not that long (plans are not so deep)
2. Some relational operators don't emit their first output until they have consumed all inputs (e.g., aggregate, sort), so we can't pipeline them
3. Often the execution cost of one operator is greater than others, so speedup gains are limited

There are multiple ways to partition data:

- Round-robin: good for sequential scans, bad for equality predicates (also called associative search, like WHERE key=value)
- Hash partitioning: good for equality predicates
- Range partitioning: good for clustering data, risks data skew though. So execution might be done in one partition (execution skew). Need to use histograms to minimize this.

Question 7.3. Describe a reason why pipeline parallelism benefits could be limited.

7.1.3 Parallelism within Relational Operators

Each relational operator has a set of input ports on which input tuples arrive and output port to which the operator's output stream is sent. There are a couple of parallelizing operators: the merge (combine several partitions from different nodes) and split (distribute data across different machines).

7.1.4 State of the Art

There are many state of the art systems. Teradata, Tandem, Gamma, Super, Bubba.

7.1.5 Future Directions

Mixing batch and OLTP queries, application program parallelism (e.g., Spark), online data reorganization, parallel query optimization (e.g., Spark).

7.2 Encapsulation of Parallelism in the Volcano Query Processing System [23]

Abstract

Volcano is a new dataflow query processing system we have developed for database systems research and education. The uniform interface between operators makes Volcano extensible by new operators. All operators are designed and coded as if they were meant for a single-process system only. When attempting to parallelize Volcano, we had to choose between two models of parallelization, called here the bracket and operator models. We describe the reasons for not choosing the bracket model, introduce the novel operator model. and provide details of Volcano's exchange operator that parallelizes all other operators. It allows intra-operator paralellism on partitioned datasets and both vertical and horizontal interoperator parallelism. The exchange operator encapsulates all parallelism issues and therefore makes implementatton of parallel database algorithms significantly easier and more robust. Included in this encapsulation is the translation between demand-driven dataflow within processes and data-driven dataflow between

processes. Since the interface between Volcano operators is similar to the one used in "real" commercial systems, the techniques described here can be used to parallelize other query processing engines.

Gamma showed us that we need an extensible query processing system that allows parallelizing of algorithms without reasoning about parallelism. How do we parallelize operators but free the programmer from reasoning about such parallelism? Volcano uses the operator model approach to parallelism, with special exchange operators. Volcano is shared-memory.

Definition 7.1. An **exchange operator** is an operator that you can insert into a query plan which automatically parallelizes the execution of the query plan. This allows you to not have to rewrite existing operators for parallelization.

7.2.1 Bracket Model of Parallelism

In the bracket model, there is a generic process template that can receive and send data and can execute exactly one operator at any point of time. The code that makes up the generic template invokes the operator (e.g., join, aggregation) and then controls execution, network, I/O on receiving and sending sides.

A problem with the bracket model is that each locus of control needs to be created, which is done by a separate schedule process. [Shreya: I don't understand this]. The bracket executes only one operator at a time and needs an external scheduler to schedule an operator. Also, tuples need to be sent over the network (which is bad for performance). So this is unsuitable for extensability.

Question 7.4. What is the bracket model of parallelism, and what is a drawback of its approach?

7.2.2 Volcano System Design

Queries are expressed as complex algebra expressions. The operators of this algebra are query processing algorithms. All algebra operators are implemented as (a tree of) iterators, i.e., they support a simple open-next-close protocol similar to conventional file scan.

Associated with each algorithm is a state record, where the arguments for the algorithm are kept. All operations on records, e.g., comparisons and hashing, are performed by support functions which are given in the state records as arguments to the iterators. This allows query processing modules to be implemented without knowledge of the internal structure of data objects.In queries with more than one operator, state records are linked together with input pointers.

Question 7.5. What is a state record, and what is its purpose?

Volcano uses "demand-driven dataflow." In NEXT, iterators return structures which contain record ids and pointers to the records which are pinned in the buffer pool. Each record pinned in the buffer is held by exactly one operator at any point in time. [Shreya: why? for complexity's sake?] After receiving a record, the operator can either hold onto it for a while (e.g., hash table) or unpin it (e.g., when a predicate fails) or pass it on to the next operator.

Question 7.6. Explain the concept of demand-driven dataflow.

7.2.3 Operator Model of Parallelization

The module responsible for parallel execution and synchronization is called the **exchange** iterator in Volcano. It is an iterator with open, next, and close procedures (same interface as other operators) so it can be placed anywhere in the query tree.

Vertical Parallelism. This is pipelining between processes. When opened, the exchange operator forks a child and establishes a region of shared memory called a port which the parent and child use to communicate with one another. The parent serves as a consumer and the child is a producer. NEXT waits for data to arrive via the port from the child. The port is like a bounded buffer that the child eagerly writes to, and the parent reads whenever its NEXT method is invoked. For flow control, the producer can decrement a semaphore and the producer can increment a semaphore. When its input is exhausted, the exchange operator in the producer/child process marks the last packet with an end-of-stream tag, passes it to the consumer, and waits until the consumer allows closing all open files (idiosyncrasy with Volcano to deal with virtual memory).

Why use eager execution for vertical parallelism (data-driven dataflow) rather than demand-driven? One, the same exchange operator should be used for horizontal parallelism,

which is easier to implement with data-driven dataflow. Second, this scheme removes the need for request messages (and associated overhead) from parent to child. They wanted something that would work well with shared-nothing architectures.

How do we alleviate back-pressure, or the problem where producers are going significantly faster than consumers? They introduce an extra flow control semaphore that producers must acquire before adding to the port. This effectively limits the size of the port at any time.

Question 7.7. Compare and contrast data-driven dataflow with demand-driven dataflow.

Question 7.8. Describe back-pressure and how Volcano handles it.

7.2.4 Horizontal Parallelism

There are 2 forms of horizontal parallelism: bushy and intra-operator parallelism. In bushy parallelism, different CPUs execute different subtrees of a complex query tree. Intra-operator paralellism means that several CPUs perform the same operator on different subsets of a stored dataset or intermediate result.

In bushy parallelism, we just need to insert one or two exchange operators into a query tree. Then multiple siblings in the query tree run in parallel.

For intra-operator parallelism, we need to partition data. Here, we can have multiple ports though. A support function is used to determine which port a packet is sent (round robin, range, or hash partition function). [Shreya: TODO: read more on this]

Question 7.9. Describe vertical, bushy, and intra-operator parallelism. How are they achieved?

7.2.5 Variants

For some operations, one could replicate or broadcast a stream to all consumers during intra-operator parallelism, instead of partitioning it. This is useful for hash division. A merge operator could sit on top of an exchange operator that parallelizes sorting.

7.3 MapReduce: simplified data processing on large clusters [16]

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/-value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system. Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues. The authors designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library.

Note 7.1. Stonebraker [2]: "Forty years of DBMS research and enterprise best practices has confirmed the position advocated by Ted Codd in 1970: Programmer and system efficiency are maximized when data management operations are coded in a high-level language and not a record-at-a-time language."

7.3.1 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines. Execution proceeds in the following steps:

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. The master picks idle workers and assigns to map or reduce tasks.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. . When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

7.3.2 Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (idle, in-progress, or completed), and the identity of the worker machine (for non-idle tasks). The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks.

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed

by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers.

The current implementation has no fault tolerance for master nodes.

If map and reduce functions are deterministic, then the computation is deterministic. The master attempts to schedule map tasks on the same machine where the data is located. Increasing M and R improves load balancing and decreases the time it takes for work to be redone after a worker failure. However, if there are too many tasks, the master becomes a memory bottleneck. They often make M=200k and R=5k, with 2k machines.

Backup tasks: One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for all sorts of reasons. So towards the end of a MR operation, the master schedules backup executions of the remaining in-progress tasks.

Question 7.10. Explain the backup optimization and why it's used.

7.3.3 Refinements

The authors discuss a number of refinements:

- User-defined partitioning functions
- Combiner functions (user-specified reduce functions to run at the mapper) to decrease state size that comes out of a mapper
- I/O types: different types have different split semantics (e.g., text at line boundaries). Users can implement their own types through a reader interface.
- Skipping bad records: sometimes there are bugs in user code that cause the map or reduce functions to crash deterministically on certain records (duh). The MR framework can inform workers to skip this task. Each worker process installs a signal handler that catches segmentation violations and bus errors and forwards a "last gasp" UDP message to the master. If the master sees multiple last gasp messages for the same record, it tells other workers not to process it.
- Local execution: MR supports a local execution to allow for easier debugging. (surprise surprise, things fail at scale)

- Status information: the master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed
- Counters: The MapReduce framework also supports global counters which can be incremented and decremented by workers and shown in the status information. Users find this helpful to "sanity check" the behavior of MR operations.

Question 7.11. Discuss 3 types of optimizations/refinements used in MapReduce.

7.4 TAG: A tiny aggregation service for ad-hoc sensor networks [39]

Abstract

We present the Tiny AGgregation (TAG) service for aggregation in low-power, distributed, wireless environments. TAG allows users to express simple, declarative queries and have them distributed and executed efficiently in networks of low-power, wireless sensors. We discuss various generic properties of aggregates, and show how those properties affect the performance of our in network approach. We include a performance study demonstrating the advantages of our approach over traditional centralized, out-of-network methods, and discuss a variety of optimizations for improving the performance and fault tolerance of the basic solution.

Users use sensors to monitor and collect data about various phenomenon and submit aggregation queries (e.g., average temperature). Previously users would run these queries in C or some other low level language. TAG is a system that allows users to express queries in SQL and distributes these queries across the network efficiently. TAG processes aggregates in the network by computing over the data as it flows through the sensors, discarding irrelevant data and combining relevant readings into more compact records when possible. TAG uses a tree-based model and bubbles up intermediate results during aggregation.

7.4.1 Motes and Ad-Hoc Networks

Motes are small battery-powered sensors with single-channel radios (can't send and receive at the same time). Motes communicate using a protocol over a shared backbone in which messages are broadcast to all motes. So any mote within hearing distance hears a message, irrespective of whether or not that mote is the intended recipient. Messages are

30 bytes, fixed size. Each device has a unique sensor ID that distinguishes it from others. All messages specify their recipient (or specify broadcast, meaning all available recipients), allowing motes to ignore messages not intended for them, although non-broadcast messages are received by all motes within range – unintended recipients simply drop messages not addressed to them.

To communicate, motes follow a routing tree. In general, TAG is agnostic to the choice of routing algorithm, requiring it to provide just two capabilities. First, it must be able to deliver query requests to all nodes in a network. Second, it must be able to provide one or more routes from every node to the root of the network where aggregation data is being collected. These routes must guarantee that at most one copy of every message arrive (no duplicates are generated).

Routing trees are ad-hoc constructed. The root broadcasts a message asking motes to organize into a routing tree; in that message it specifies its own id and its level, or distance from the root (in this case, zero.) Any mote without an assigned level that hears this message assigns its own level to be the level in the message plus one. It also chooses the sender of the message as its parent, through which it will route messages to the root. During aggregation, children send messages up the tree to the root.

7.4.2 Query Model and Environment

They support SQL-style queries without joins over a single table called sensors, whose schema is known at the base. This table can be thought of as an append-only relational table with one attribute per input of the motes (e.g., temperature, light.)

Queries are executed once every epoch, to return *continuous* results to a user. Users can deregister their queries at any time. So TAG queries deviate from SQL queries only in that they are streaming. But the epoch must be enough time for aggregates to be computed by the network.

The approach used in such systems (and followed in TAG) is to implement aggregation via three functions: a merging function f, an initializer i, and an evaluator e. The initializer is needed to specify how to instantiate a state record for a single sensor v—for unweighted average, it might be the tuple $i(x) = \langle x, 1 \rangle$. Finally, the evaluator takes a partial state record and computes the actual value of the aggregation. ALl in all; leaves generate intermediate records with i which they send to their parents. Inner nodes merge intermediate records with f to create new intermediate records. Running e on the final intermediate records produces the final aggregate.

Question 7.12. Explain the query model and structure of aggregates in TAG. For instance, what functions does TAG use to deal with aggregation?

Taxonomy of Aggregates. They did not want to restrict the user to only SQL aggregates (the 5 count, min, max, sum, average). They classify aggregates according to four properties that are particularly important to sensor networks:

1. Duplicate sensitivity: aggregates unaffected by duplicate readings, e.g., count, sum, average, median, histogram
2. Exemplary vs summary (tolerance of loss): exemplary returns one or more records (max, min, median), summary returns something on all the records (count, sum, average, count distinct, histogram)
3. Monotonic: when 2 partial states s_1 and s_2 are combined via f, the resulting state record s' will have property such that $\forall s_1, s_2, e(s') \geq \max(e(s_1), e(s_2))$ or $\forall s_1, s_2, e(s') \leq \min(e(s_1), e(s_2))$. This allows us to perform early predicate evaluation and save messages describing partial state. Ex: max, min, count, sum, count distinct. Think of this as intermediate results are always increasing or decreasing.
4. Partial state: this relates to the amount of state required as a relation to the query. There are 3 categories this could take on: distributive (same size as final aggregate, like max/min/count/sum), algebraic (constant size like average), holistic (size proportional to the partition computed on, like median), unique (size proportional to number of distinct values, like count distinct), content-sensitive (size proportional to a statistical property, like histogram).

Question 7.13. What are the 4 properties of aggregates introduced in the paper?

Motes also maintain a catalog of their available attributes which is cached by the querier. A mote returns NULL if it is queried for an attribute it doesn't have. The central query processor caches the set of attributes which can be queried.

7.4.3 In Network Aggregate

TAG consists of two phases: a distribution phase, in which aggregate queries are pushed down into the network, and a collection phase, where the aggregate values are continually routed up from children to parents.

When the root receives a query with epoch duration d, it propagates the query down through the tree. Whenever a parent forwards the query down towards its children, it includes a deadline by which it expects its children to return a response. When a child receives such a message, it decreases the deadline before propagating to its children. What

is the deadline? If all nodes know the depth of the network, then they can set the depth as a heuristic. The paper leaves pipelining messages for future work.

For GROUP BY queries, nodes annotate every intermediate record with the group that it belongs to. Nodes maintain a table mapping group ids to intermediate records. If the node runs out of memory, it evicts some of these entries upwards through the tree. Because groups can be evicted, the base station at the top of the network may be called upon to combine partial groups to form an accurate aggregate value. Evicting partially computed groups is known as partial preaggregation. Monotonic aggregates with a HAVING clause can be pruned early.

The principal advantage of TAG is its ability to dramatically decrease the communication required to compute an aggregate versus a centralized aggregation approach. Another advantage is the ability to tolerate loss, through partial state records. Another advantage is through the limit of one message per epoch, nodes can go into deep sleeps and not lose power.

Question 7.14. Discuss some advantages of TAG over a centralized aggregation approach.

7.4.4 Optimizations

They employed several optimizations:

- when a node misses an initial request to begin aggregation: it can initiate aggregation even after missing the start request by snooping on the network traffic of nearby nodes. For example, if a node snoops the MAX value sent by another node and it is higher than its current value, then it does not need to propagate its value upwards.
- Motes can propagate information down the tree that motes can use to prune some messages. The paper describes this strategy as hypothesis testing. For certain classes of aggregates, if a node is presented with a guess as to the proper value of an aggregate, it can decide locally whether contributing its reading and the readings of its children will affect the value of the aggregate. For example, imagine the root sends an estimated average and an error bounds. If a node's intermediate average is within the error bounds, it does not have to send it upwards.

In order to tolerate faults, nodes automatically reorganize themselves in the face of mote failure. Every mote measures the link quality to all of its neighbors (the proportion of

packets received from each neighbor). When a node n observes that the link quality to its parent p is significantly worse than that of some other node p', it chooses p' as its new parent if p' is as close or closer to the root as p and p' does not believe n is its parent (avoiding cycles).

Second, when a node observes that it has not heard from its parent for some fixed period of time (relative to the epoch duration of the query it is currently running), it assumes its parent has failed or moved away. It resets its local level to ∞ and picks a new parent according to the link quality metric. Child nodes must reselect their parent when they observe that their own parent's level has gone up.

Question 7.15. What is hypothesis testing? Snooping?

Question 7.16. Describe fault tolerance in TAG.

7.4.5 Loss

Intermediate nodes can cache the values of their children and relay them if their children have failed. Though, this uses up RAM that could otherwise be used for the grouping cache. For duplicate insensitive algorithms, motes can broadcast their values to all parents. For things like SUM, a mote can send half of its value to two of its parents.

7.5 Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing [53]

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarsegrained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel,

and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

Frameworks like MapReduce made processing large amounts of data easier, but they did not leverage distributed memory. MR writes all intermediate state to disk, making it unacceptably slow for iterative and interactive workflows (e.g., k-means, ad-hoc queries). Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse—Pregel, HaLoop.

This paper proposes a new abstraction, RDDs, that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge for designing RDDs s is defining a programming interface that can provide fault tolerance efficiently. RDDs are immutable partitioned collections of records. Unlike pure distributed shared memory abstractions which allow for arbitrary fine-grained writes, RDDs can only be constructed using coarse-grained transformations from on-disk data or other RDDs.

This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data. If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. Thus, lost data can be recovered, often quite quickly, without requiring costly replication.

Note 7.2. Formally, an RDD is a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs. Such operations are called transformations—like map, filter, and join.

RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage. This is a powerful property: in essence, a program cannot reference an RDD that it cannot reconstruct after a failure.

Spark has a Scala-integrated API and comes with a modified interactive interpreter. It also includes a large number of useful transformations (which construct RDDs) and actions (which derive data from RDDs). Users can also manually specify RDD persistence and partitioning to further improve performance.

7.5.1 Representing RDDs

The most interesting question in designing this interface is how to represent dependencies between RDDs. We found it both sufficient and useful to classify dependencies into two types: narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD, wide dependencies, where multiple child partitions may depend on it. For example, map leads to a narrow dependency, while join leads to to wide dependencies (unless the parents are hash-partitioned).

This distinction is useful for two reasons. First, narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis. In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce like operation.

Second, recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution.

For wide dependencies (i.e., shuffle dependencies), we currently materialize intermediate records on the nodes holding parent partitions to simplify fault recovery, much like MapReduce materializes map outputs.

Question 7.17. Discuss the difference between narrow and wide dependencies. What is the optimization used for simplified fault recovery for wide dependencies?

7.5.2 Memory Management

To manage the limited memory available, we use an LRU eviction policy at the level of RDDs. When a new RDD partition is computed but there is not enough space to store it, we evict a partition from the least recently accessed RDD, unless this is the same RDD as the one with the new partition. In that case, we keep the old partition in memory to prevent cycling partitions from the same RDD in and out. This is important because most operations will run tasks over an entire RDD, so it is quite likely that the partition already in memory will be needed in the future. We found this default policy to work well in all our applications so far, but we also give users further control via a "persistence priority" for each RDD.

Question 7.18. Discuss the memory management scheme for Spark.

8 The Web and Databases

We cover:

- Combining Systems and Databases: A Search Engine Retrospective
- The Anatomy of a Large-Scale Hypertextual Web Search Engine
- WebTables: Exploring the Power of Tables on the Web

8.1 Combining Systems and Databases: A Search Engine Retrospective [10]

Abstract

Although a search engine manages a great deal of data and responds to queries, it is not accurately described as a "database" or DMBS. We believe that it represents the first of many application-specific data systems built by the systems community that must exploit the principles of databases without necessarily using the (current) database implementations. In this paper, we present how a search engine should have been designed in hindsight. Although much of the material has not been presented before, the contribution is not in the specific design, but rather in the combination of principles from the largely independent disciplines of "systems" and "databases." Thus we present the design using the ideas and vocabulary of the database community as a model of how to design data-intensive systems. We then draw some conclusions about the application of database principles to other "out of the box" data-intensive systems.

Search engines (SEs) are arguably the largest data management systems in the world; although there are larger databases in total storage there is nothing close in query volume. Despite the size and complexity of these systems, they make almost no use of DBMS systems. Principles of databases that it should have followed are:

- Top-down design: start with the desired semantics & develops mechanisms to implement those semantics
- Data independence: data should exist in sets without pointers, allowing evolution of storage and representation and simplifying recovery and fault tolerance
- Declarative query language: logical query plan that is separate from physical query plan. But ACID transactions are not the right semantics for search engines.

8.1.1 SE Overview

A crawler surfs the web, the indexer builds inverted indexes and scores documents, and the server parses, optimizes, and executes queries. This is done offline. In general, the goal is to move work from the (online) web servers to the indexer, so that the servers have the absolute minimum amount of work to do per query.

Conceptually, a query defines some words and properties that a matching document should or should not contain. A document is normally a web page, but could also be a news article or an e-mail message. Documents contain words and have properties. Words have scores (for the document), and properties are boolean (present or absent in the doc, e.g., language:english). Queries return documents with the given words that satisfy the given properties. Search results are scored based on document quality and the score of the words within the document (as determined by the indexer).

The score for a query $Q = \{w_1, w_2, \ldots, w_k\}$ is:

$$\text{Score}(Q, d) = \text{Quality}(d) + \sum_{i}^{k} \text{Score}(w_i, d)$$

The quality term is independent of the query words and reflects things like length (shorter is generally better), popularity, incoming links, quality of the containing site, and external reviews. I guess this is a user-defined function.

8.1.2 Logical Query Plan

The docs and inverted indexes are stored as relational tables with the following schema:

```
Document(DocID, URL, Date, Size, Abstract), 3B rows
Word(WordID, DocID, Score, Position Info), 1T rows
Property(WordID, DocID), 1B rows
Term(String, WordID, Stats), 10M rows
```

The word and property tables are tables are inverted indexes from words and properties to documents. The user queries a word, which gets mapped using the term table to a word ID and some stats/properties, which get used for selectivity. The query plan is as in Figure 5.

Applying the top-down principle, we design a small query language for this application, rather than using SQL. Here is the BNF for one possible query language:

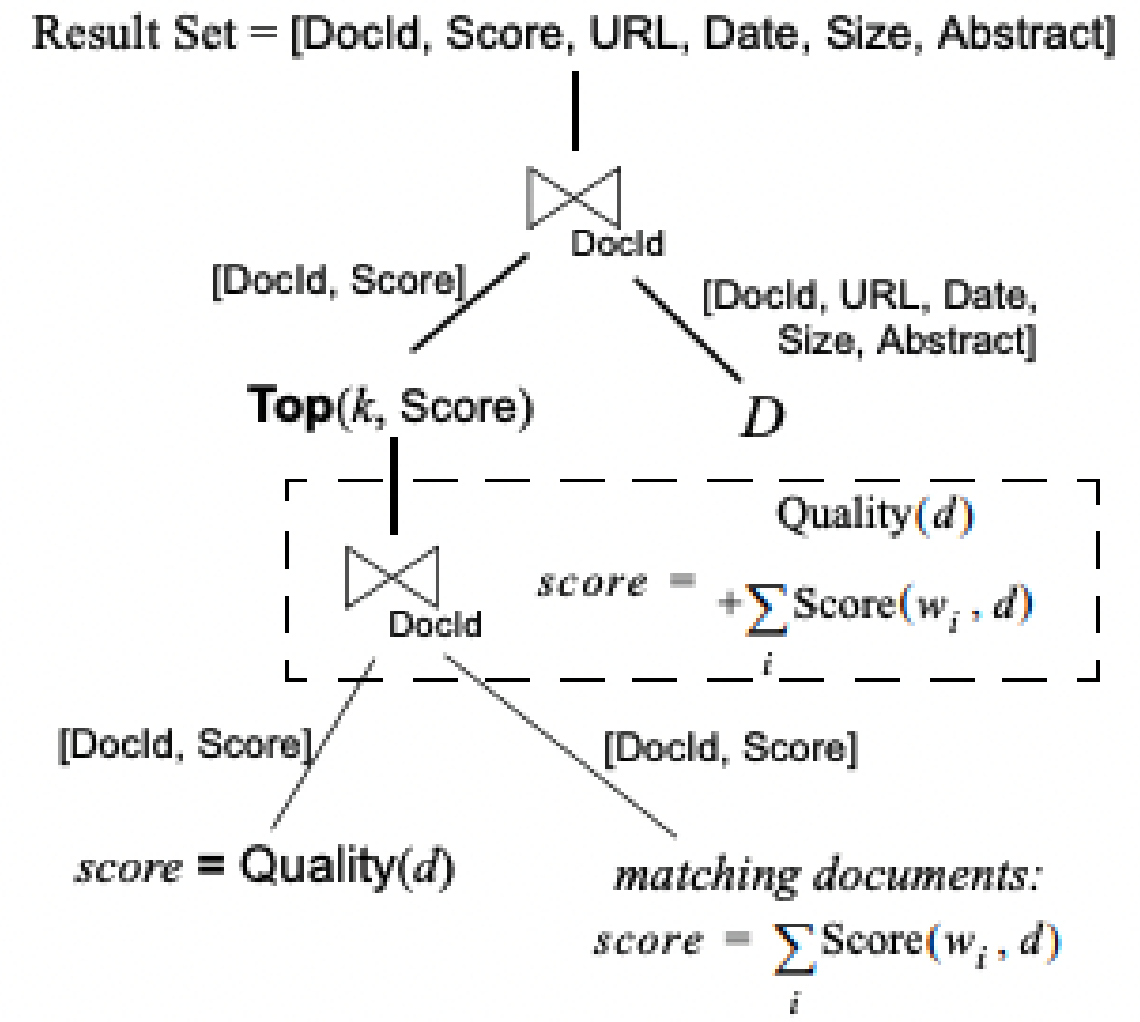

Figure 2: The General Query Plan
After finding the set of matching documents and their scores, the **Top** operator passes up the top *k* results (in order) to an equijoin that adds in the document information.

Figure 5: Search engine query plan

```
expr: expr AND expr
| expr OR expr
| expr FILTER prop
| word

prop: prop AND prop
| prop OR prop
| NOT prop
| NOT expr
| property
```

So we can have a query like: `san AND francisco; (bay AND area) FILTER lang:english`

8.1.3 Query Implementation

There is really only one kind of access method: sequential scan of a sorted inverted index, which is just a sorted list of all of the documents that contain a given term. An unusual aspect of the physical plan is that we cache all of the intermediate values (for use by other queries), and do not pipeline the plan.

Given that we keep all intermediate results, there is no space savings for pipelining. Pipelining could still be used to reduce query latency, but we care more about throughput than latency, and throughput is higher without pipelining, due to lower per-tuple overhead and better (memory) cache behavior.

Question 8.1. Why wouldn't we use pipelining in building a DBMS for search engines?

The query optimizer transforms logical queries into physical query plans using these four operators: or (returns expr), or (returns property), and (inner join returns property), filter (inner join returns expr). Since data is sorted, joins are performed multi-way.

One nice property of using multiway joins is that it mitigates the need for estimating selectivity. Selectivity estimation is normally needed to compute the size of an input to another operator; increasing the fan in of an (inner) join limits the work to the actual size of the smallest input and thus decreases the need for estimates.

The idea here is to flatten the tree to use fewer but wider joins. First, flatten all OR and AND chains. Then, looked for cached subexpressions. Basically, the optimizer prefers flatter query plans with multiway pre-sorted merge joins on inverted indexes.

Question 8.2. Explain how the cache is used in query optimization for SE.

8.1.4 Implementation on a cluster

The bulk of every query goes to every node and executes the same code on different data. From the database perspective, this means a mixture of replication for small tables and horizontal fragmentation (also known as "range partitioning") for large tables.

Question 8.3. Why would we horizontally partition the document, word, and property tables?

Initially, using a load balancer, a query is routed to exactly one node, called the master for that query. Most nodes will be the master for some queries and a follower for the others. The master node computes and optimizes the query plan, issues the query to all other nodes (the followers), and collates the results. Each follower computes its top k results, and the master then computes the top k overall. Finally, the last equijoin with the document table, D, is done via a distributed hash join with one lookup for each of the k results (which may also be cached locally)—dose without any repartitioning of the table.

Using the master to compute the query plan has some issues. The advantage is that the plan is computed only once. The disadvantage is that cache contents in each node are not the same, so a follower might have to recompute something that the master expected to be in the cache.

Question 8.4. What is an advantage and disadvantage of using a master node to compute the query plan?

One of the most important optimizations is compression of inverted indices. Since there are usually no individual updates (we only update the entire table), we can keep the table compressed. Also, there is no random access—they are always fully scanned.

Additionally, we can use the A* algorithm to speed things up. The master node computes its top k, and uses the kth score as an estimate on the lower bound of scores, that follower nodes can use to prune out subqueries that cannot beat this lower bound. For example, for a term in a multiway join, there is typically some score below which you need not

perform the join, since even with the best values for the other terms the end score will not make the top k.

Question 8.5. Why can we use compression for SE DBMSes?

Question 8.6. How is the A* algorithm used in pruning subqueries in the search?

8.1.5 Updates

The document and indexes relations are divided into chunks. The indexer and crawler create updates and insertions on the granularity of a chunk for atomicity. Only whole tables are updated, and not individual rows. Updates should be atomic with respect to queries; that is, updates always occur between queries.

The simplest implementation of caching uses a separate cache for each chunk, in which case we can just invalidate the whole cache for that chunk.

8.1.6 Fault Tolerance

Whenever a node fails, the query can be retried elsewhere. A search engine can gracefully degrade under heavy load by only searching a subset of the chunks or by outright rejecting expensive queries.

8.1.7 Other Topics

- Personalization: Search engines can store user personalization information in a cookie, or it can store a user id in the cookie and store the information in a database. Storing the information in a database makes it much easier to do things like schema evolution.
- Logging: Search engines produce a lot of logs and require custom logging frameworks.
- Query rewriting: Sometimes queries can be rewritten with things like the preferred language or some words on the current page being looked at.

- Phrase queries: In order to support phrase queries (e.g. "New York"), search engines can measure the nearness of words in a document or can straight up search for whole sequences of words.

8.2 The Anatomy of a Large-Scale Hypertextual Web Search Engine [11]

Abstract

In this paper, we present Google, a prototype of a large-scale search engine which makes heavy use of the structure present in hypertext. Google is designed to crawl and index the Web efficiently and produce much more satisfying search results than existing systems. The prototype with a full text and hyperlink database of at least 24 million pages is available at http://google.stanford.edu/.

To engineer a search engine is a challenging task. Search engines index tens to hundreds of millions of web pages involving a comparable number of distinct terms. They answer tens of millions of queries every day. Despite the importance of large-scale search engines on the web, very little academic research has been done on them. Furthermore, due to rapid advance in technology and web proliferation, creating a web search engine today is very different from three years ago. This paper provides an in-depth description of our large-scale web search engine – the first such detailed public description we know of to date.

Apart from the problems of scaling traditional search techniques to data of this magnitude, there are new technical challenges involved with using the additional information present in hypertext to produce better search results. This paper addresses this question of how to build a practical large-scale system which can exploit the additional information present in hypertext. Also we look at the problem of how to effectively deal with uncontrolled hypertext collections where anyone can publish anything they want.

Design goals were: improved search quality, and build systems that lots of people cal use. Google stores all of the actual documents it crawls in compressed form. One of our main goals in designing Google was to set up an environment where other researchers can come in quickly, process large chunks of the web, and produce interesting results that would have been very difficult to produce otherwise.

8.2.1 System Features

For page A with links $T_1, \ldots, T_n$ that point to it, d a damping factor, and $C(A)$ number of links going out of A, we get PageRank:

$$\mathrm{PR}(A) = (1-d) + d(\mathrm{PR}(T_1)/C(T_1) + \ldots + \mathrm{PR}(T_n)/C(T_n))$$

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one. PageRank can be thought of as a model of user behavior. We assume there is a "random surfer" who is given a web page at random and keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page. The probability that the random surfer visits a page is its PageRank. And, the d damping factor is the probability at each page the "random surfer" will get bored and request another random page.

Google does other smart feature engineering. Most search engines associate the text of a link with the page that the link is on. In addition, they associate it with the page the link points to (anchor text). Also, they have location information for all hits and weight fonts that are larger/bolder higher.

8.2.2 Anatomy

A URL Server sends URLs to a set of crawlers, which download sites and send them to a store server which compresses them and stores them in a repository. An indexer uses the repository to build a forward index (stored in a set of barrels, partially-sorted) which maps documents to a set of hits. Each hit records a word in the document, its position in the document, its boldness, etc. The indexer also stores all anchors in a file which a URL resolver uses to build a graph for PageRank. A sorter periodically converts forward indexes into inverted indexes.

The data structures are:

- BigFiles: virtual files spanning multiple filesystems
- Repository: full HTML of every page, compressed as fast as possible. ALl data structures can be built from this and a file that lists crawler errors.
- Document index: sequential access for doc IDs that points to the repository, URL, and site title. It also maps URLs to docIDs.
- Lexicon: kept in memory, list of words
- Hit list: compressed list of occurrences of a particular word in a document including position, font, and capitalization information. This accounts for most of the space in the forward and backward indices.
- Forward index: list of doc ids followed by word id and series of hits. Horizontally partitioned by wordID across barrels.
- Inverted index: maps words to lists of docIDs. There are two ways to sort docIDs:one sorting docIDs by occurrence of word in each document and another sorting docIDs by docID. The latter allows for quick merging; the former gives low

latency one word queries. They chose a compromise between these options, keeping two sets of inverted barrels—one set for hit lists which include title or anchor hits and another set for all hit lists. This way, they check the first set of barrels first and if there are not enough matches within those barrels they check the larger ones.

Question 8.7. Describe the inverted index.

8.2.3 Crawling

The URL server and the crawlers are implemented in Python. Each crawler has about 300 open connections, performs event loop asynchronous IO, and maintains a local DNS cache to boost performance. Crawling is the most fragile part of the entire search engine and requires a lot of corner case handling. It is 100 web pages/s at peak speed.

8.2.4 Indexing

They wrote a custom HTML parser. After each document is parsed, it is encoded into a number of barrels. Every word is converted into a wordID by using an in-memory hash table – the lexicon. New additions to the lexicon hash table are logged to a file and added in batch later. The main difficulty with parallelization of the indexing phase is that the lexicon needs to be shared. Instead of sharing the lexicon, we took the approach of writing a log of all the extra words that were not in a base lexicon, which we fixed at 14 million words. That way multiple indexers can run in parallel and then the small log file of extra words can be processed by one final indexer.

Question 8.8. How did they parallelize indexing, especially with new additions to the lexicon?

8.2.5 Search

They do the following steps:

1. Parse query
2. Convert words into wordIDs

3. Seek to start of doclist in short backwards index for every word
4. Scan through doclists until there is a document that matches all search terms
5. For each resulting document, compute the rank for that query
6. If we are in the short barrels and at the end of any doclist, seek to the start of the doclist in the full barrel for every word and go to step 4.
7. If we are not at the end of any doclist go to step 4.
8. Sort the documents that have matched by rank and return the top k.

For single word queries, documents are ranked based on a lot of factors including the frequency of the words, the boldness of the words, PageRank, etc. For multiword queries, the words are scored as before but additionally with the proximity of the words being taken into account. The proximity is based on how far apart the hits are in the document (or anchor) but is classified into 10 different value ”bins” ranging from a phrase match to ”not even close”.

8.3 WebTables: Exploring the Power of Tables on the Web [12]

Abstract

The World-Wide Web consists of a huge number of unstructured documents, but it also contains structured data in the form of HTML tables. We extracted 14.1 billion HTML tables from Google's general-purpose web crawl, and used statistical classification techniques to find the estimated 154M that contain high-quality relational data. Because each relational table has its own "schema" of labeled and typed columns, each such table can be considered a small structured database. The resulting corpus of databases is larger than any other corpus we are aware of, by at least five orders of magnitude. We describe the WebTables system to explore two fundamental questions about this collection of databases. First, what are effective techniques for searching for structured data at search-engine scales? Second, what additional power can be derived by analyzing such a huge corpus? First, we develop new techniques for keyword search over a corpus of tables, and show that they can achieve substantially higher relevance than solutions based on a traditional search engine. Second, we introduce a new object derived from the database corpus: the attribute correlation statistics database (AcsDB) that records corpus-wide statistics on cooccurrences of schema elements. In addition to improving search relevance, the AcsDB makes possible several novel applications: schema auto-complete, which helps a database designer to choose schema elements; attribute synonym finding, which automatically computes attribute synonym pairs for schema matching; and join-graph traversal, which allows a user to navigate between extracted schemas using automatically-generated join links.

They scrape a bunch of tables from Google (14.1 billion!) and get 154M well formed, nice relational tables. To perform relation ranking, i.e., to sort relations by relevance in response to a user's keyword search query, WebTables must solve the new problem of ranking millions of individual databases, each with a separate schema and set of tuples.

They have an attribute correlations statistics database (ACSDb), which is a set of statistics about schemas in the corpus. The ACSDb can be used to compute the likelihood of a certain attribute appearing in a schema or the likelihood of a certain attribute appearing in a schema conditioned on the fact that some other attribute appears.

8.3.1 Relations

We tuned the relation-filter for relatively high recall and low precision, so that we lose relatively few true relations. For each relation $R \in \mathcal{R}$, the corpus of DBs, they have the following:

- URL R_u and offset R_i within the page R was extracted. These are the unique identifiers
- Schema R_S, an ordered list of attribute labels (can be empty strings)
- A list of tuples R_T where tuple t is a list of data strings (can be empty strings in the list)

The ACSDb lists each unique S found in the set of relations, along with a count that indicates how many relations contain the given S. After removing all attributes and all schemas that appear only once in the entire extracted relation corpus, they computed an ACSDb with 5.4M unique attribute names and 2.6M unique schemas.

8.3.2 Relation Search

When a user enters a query like "population of US capitals", we want to return a ranked list of tables from our corpus. To do so, the authors implemented six ranking functions. Each ranking function is parameterized by a search query q and a limit k on the number of results.

- naiveRank: sends query to Google. extracts and returns all tables in the top k results
- filterRank: sends query to Google. extracts top k relations to return, with same page results as naiveRank

- featureRank: ranks each document using a classifier trained on a number of hand-scored documents. The classifier uses features such as the number of rows, the number of columns, the number of nulls, the number of hits in the header, the number of hits in the leftmost column, etc. The two most heavily-weighted features for the estimator are the number of hits in each relation's schema, and the number of hits in each relation's leftmost column.
- schemaRank: ranks each relation using the featureRank classifier and a schema coherency metric, which measures how well attributes of a schema fit together. They use point-wise mutual information score, which is log joint over the product of the marginals. This value is large when the attributes are correlated, zero when independent, and negative when negatively correlated.

8.3.3 Indexing

The inverted index is a structure that maps each term to a sorted posting list of (docid, offset) pairs that describe each occurrence of the term in the corpus. The offset is used to compute adjacencies.

Unlike the "linear text" model that a single offset value implies, WebTables data exists in two dimensions, and the ranking function uses both the horizontal and vertical offsets to compute the input scoring features. Thus, we adorn each element in the posting list with a two-dimensional (x, y) offset that describes where in the table the search term can be found. Using this offset WebTables can compute, for example, whether a single posting is in the leftmost column, or the top row, or both. So, To serve queries, we maintain an inverted index which maps a word w to a list of pairs of the form $(r, (x, y))$ where w appears in relation r at position (x, y).

Question 8.9. Explain how the inverted index used must be changed from the original PageRank scheme to be used by WebTables.

8.3.4 ACSDb Applications

They describe a few applications:

- Schema autocomplete: The autocomplete works by incrementally suggesting the most likely attributed conditioned on the existing attributes and the existing suggested attributes.

- Attribute Synonym-Finding: the same attribute is often named different things in different relations. Given a set of words C, the synonym finder suggests pairs of synonyms that are likely in the context of words C. It does so with two observations. First, if two words are synonyms then the likelihood that they both appear in the same schema $p(a, b)$ should be very low. Second, $p(z|a, C)$ should be roughly equal to $p(z|b, C)$. That is, conditioning on a or conditioning on b should have a similar effect.

- Join Graph Traversal: they construct a join graph by creating a node for each unique schema, and an undirected join link between any two schemas that share a label. This graph is really big, so navigating it becomes complex. So they cluster neighboring nodes together if they are similar, based on some similarity score for a shared attribute (seeing if the shared attribute plays a similar role in each schema). This is similar to schema coherency.

9 Materialized Views, Cubes and Aggregation

We cover the following:

- Materialized Views
- On the Computation of Multidimensional Aggregates
- Implementing Data Cubes Efficiently
- Informix Under Control: Online Query Processing
- BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

9.1 Materialized Views [14]

[Shreya: TODO]

9.2 On the Computation of Multidimensional Aggregates [29]

At the heart of all OLAP or multidimensional data analysis applications is the ability to simultaneously aggregate across many sets of dimensions. Computing multidimensional aggregates is a performance bottleneck for these applications. This paper presents fast algorithms for computing a collection of groupbys. We focus on a special case of the aggregation problem—computation of the CUBE operator. The CUBE operator requires computing group-bys on all possible combinations of a list of attributes, and is equivalent to the union of a number of standard group-by operations. We show how the structure of CUBE computation can be viewed in terms of a hierarchy of group-by operations. Our algorithms extend sort-based and hash-based grouping methods with several optimizations, like combining common operations across multiple group-bys, caching, and using pre-computed group-bys for computing other group-bys. Empirical evaluation shows that the resulting algorithms give much better performance compared to straightforward methods.

Data analysts who interact with OLAP databases issue a lot of GROUP BY queries over huge amounts of data. These aggregates are often too big to compute interactively, so data analysts to resort to computing them ahead of time.

The CUBE operator builds a family of aggregates all at once, for all subsets of the defined columns. This paper presents a heuristic-based algorithm to efficiently compute a cube. It also proves that finding an optimal cube is NP-hard.

9.2.1 Optimizations Possible

We assume that aggregate functions are homomorphic (e.g., sum but not median).

- Smallest-parent: compute a group-by from the smallest previously computed group-by that is a superset
- Cache-results: cache results of a groupby in memory
- Amortize scans: computing as many groupbys as possible together in memory, in a scan
- Share sorts: share sorting cost across multiple groupbys
- Share partitions: specific to hash-based

9.2.2 Multiple Independent Group-By Queries (Independent Method)

We can compute and materialize the base cuboid and then independently compute every other cuboid from the base cuboid. We can use a standard sort based or hash based group-by implementation.

9.2.3 Hierarchy of Group-By Queries (Parent Method)

We can compute a cuboid on attributes using a cuboid previously computed with the smallest superset.

9.2.4 Overlap Method

We use the parent method but concurrently compute multiple cuboids at once. This method is the focus of the paper. This paper only discusses how to compute cuboids using sorting. Sorting is desirable because a cuboid derived from a sorted cuboid parent is itself sorted. Hash based methods exist too but are not discussed here.

Consider a cuboid A, B, C and we want to compute A, C. Note that A, B, C will be in sorted order by A, B, C (e.g., ascending). Each distinct value of A is a partition. Each sorted run is the first k tuples of a partition in A, B, C such that you get all distinct values of A, C.

Question 9.1. Describe the difference between sorted runs and partitions.

To minimize partition size, we prefer parents whose dropped attributes are furthest to the right. For example, to compute (B, C), we prefer (B, C, D) to (A, B, C). If a cuboid is evaluated partition-by-partition, then some of its children can be evaluated concurrently. Given a limited amount of memory, we need to choose which cuboids to evaluate in parallel. Finding an optimum schedule is NP-hard, but an eager approach walks breadth first left-to-right down the subset lattice.

9.2.5 Some Important Issues

Incorrect estimation: partition sizes may have been estimated incorrectly. At runtime, the memory allocated to each cuboid can be adjusted dynamically.

Limiting the number of sorted runs: to limit the number of sorted run files, we can append sorted runs from one partition onto sorted runs from previous partitions. We need only limit the number of distinct values of the parent's dropped column.

Choosing an initial sort order: We want smaller cuboids to be on the right since they have bigger partitions. Or, we can put the attributes with the fewest number of distinct values on the right to limit the number of sorted runs.

Question 9.2. What attributes would we want to put on the right? More or less distinct values? Why?

9.3 Implementing Data Cubes Efficiently [30]

Decision support applications involve complex queries on very large databases. Since response times should be small, query optimization is critical. Users typically view the data as multidimensional data cubes. Each cell of the data cube is a view consisting of an aggregation of interest, like total sales. The values of many of these cells are dependent on the values of other cells in the data cube. .A common and powerful query optimization technique is to materialize some or all of these cells rather than compute them from raw data each time. Commercial systems differ mainly in their approach to materializing the data cube. In this paper, we investigate the issue of which cells (views) to materialize

when it is too expensive to materialize all views. A lattice framework is used to express dependencies among views. We present greedy algorithms that work off this lattice and determine a good set of views to materialize. The greedy algorithm performs within a small constant factor of optimal under a variety of models. We then consider the most common case of the hypercube lattice and examine the choice of materialized views for hypercubes in detail, giving some good tradeoffs between the space used and the average time to answer a query.

On one extreme, we can materialize the entire cube and speed up every query. On the other extreme, we can materialize none of the cube and run all queries from scratch. This paper presents an algorithm which selects an optimal amount of the cube to pre-materialize.

Imagine performing a GROUP BY on a set of attributes A, B, and C. We can imagine the result of the GROUP BY as a three-dimensional cube in which one axis is the domain of A, the next axis is the domain of B, and the final axis is the domain of C. Each entry of the cube contains the aggregated value. We can also introduce a new value ALL into each domain which grows the cube to include aggregates along other dimensions. The cube (without ALL) corresponds to a GROUP BY on all attributes. Every face of the cube (view, query, cuboid) corresponds to a GROUP BY on some subset of the attributes.

9.3.1 The Lattice Framework

$Q_1 \preceq Q_2$ if Q_1 can be answered using only the results of Q_2. Sets of attributes naturally form a set lattice where, for example $(A, B) \preceq (A, B, C)$. Moreover, assume that each attribute forms a hierarchy. For example, a time attribute can be divided into days, weeks, and years; for example, $(none) \preceq (year) \preceq (month) \preceq (day)$. These hierarchies are used for drill-downs and roll-ups. Drill-down is the process of viewing data at progressively more detailed levels. For example. a user drills down by first looking at the total sales per year and then total sales per month and finally, sales on a given day. Roll-up is just the opposite: it is the process of viewing data in progressively less detail.

We assume each attribute has some hierarchy and model a query as a product lattice of all hierarchy lattices.

Question 9.3. What are drill-downs and roll-ups in querying operations?

9.3.2 The Cost Model

Let $\langle L, \preceq \rangle$ be a lattice of queries (views). To answer query Q, we need a materialized query Q_A where $Q \leq Q_A$. We assume that there are no indexes on the cuboids, so the time to query any cuboid takes time proportional to $|Q_A|$ (number of rows present in the table for the Q_A).

The sizes of cuboids can be estimated with sampling, or by sampling the number of unique values of each attributes and multiplying them together assuming independence.

9.3.3 Optimizing Data-Cube Lattices

We want to find an optimal point in the time-space tradeoff of materializing parts of a datacube, but what is optimal? Here, we consider a simple problem and later relax the assumptions. We want to minimize the time taken to run all queries, subject to a fixed number of materialized views (rather than fixed amount of memory).

Here is a greedy approximation: Let each view v have a cost $C(v)$ which is the number of tuples in v. First, let's define the benefit $B(v, S)$ of view s given existing materialized views S. For every $w \leq v$, let u be the lowest cost view in S such that $w \leq u$. If $C(u) > C(v)$, let $B_w = C(u) - C(v)$. Otherwise, let $B_w = 0$. $B(v, S) = \sum_{w \leq v} B_w$. Given a limit of k materialized views, the greedy algorithm incrementally picks the k with maximum benefit.

The benefit of the greedy approach compared to the benefit of the optimal approach is tightly lower bounded by $1 - \frac{1}{e}$. We can relax our assumptions a bit in two ways. First, we can assume that some views are run more than others. To accommodate this, we weight benefit by the likelihood of view w. Second, we can assume a fixed space rather than a fixed number of views by measuring benefit in terms of benefit per space instead of overall benefit. Given some slight caveats, all bounds still apply.

9.3.4 The Hypercube Lattice

When attributes do not have any hierarchies, we get a hypercube lattice. Assume each attribute has the same domain with size r and assume the top lattice has m values. A cuboid with i attributes has approximately r^i possible entries. If $r^i > m$, then it only really has m entries. Otherwise it has m. If $r^i > m$, then there is no reason to materialize the view. The paper describes the space and time for various parameters of r and m.

9.4 Informix Under Control: Online Query Processing [32]

Abstract

The goal of the CONTROL project at Berkeley is to develop systems for interactive analysis of large data sets. We focus on systems that provide users with iteratively refining answers to requests and online control of processing, thereby tightening the loop in the data analysis process. This paper presents the database-centric subproject of CONTROL: a complete online query processing facility, implemented in a commercial Object-Relational DBMS from Informix. We describe the algorithms at the core of the system, and detail the end-to-end issues required to bring the algorithms together and deliver a complete system.

When you submit a SQL query to a traditional relational database, it runs for a while, you sit there waiting patiently, and then it spits out an answer. For big data or complicated queries, you might be sitting and waiting patiently for quite some time. This paper focuses on how to implement a database which streams and iteratively refines output to give users more insight into what's going on.

9.4.1 Application scenarios and performance requirements

Online aggregation: users issue a group-by query like SELECT college, AVG(grade) FROM enroll GROUP BY college and see an iteratively refined confidence interval for each college. Controls can speed up or slow down the processing of certain groups.

Online enumeration: users interact with a spreadsheet that lazily streams in data values from a database.

Online data visualization: visualizations, like showing a heat map of field goal percentage on a basketball court, can involve streaming in data lazily and aggregating it.

Online data mining: e.g., frequent itemset. To use an association rule application, a user specifies values for two variables: one that sets a minimum threshold on the amount of evidence required for a set of items to be produced (minsupport) and another which sets a minimum threshold on the correlation between the items in the set (minconfidence).). These algorithms can run for hours without output, before producing association rules that passed the minimum support and confidence thresholds.

9.4.2 Randomized Data Access and Physical Database Design

Things like online aggregation require us to draw random samples of the data (otherwise the confidence bars would be all out of whack). Informix stores data randomly on disk by clustering on a pseudo-random pseudo-key. To insert a tuple into the randomly shuffled data, we randomly replace an existing tuple and append the replaced tuple. Repeatedly scanning the shuffled data can lead to statistical anomalies, so we start at randomly selected offsets or re-shuffle the data every once in a while. We'd need to keep the underlying data sorted in a more sane way.

9.4.3 Preferential data delivery: Online reordering

A key aspect of an online query processing system is that users perceive data being processed over time. Hence an important performance goal for these systems is to present data of interest early on in the processing, so that users can get satisfactory results quickly, halt processing early, and move on to their next request.

A re-order operator can pre-fetch tuples from disk and output the more "interesting" tuples output, spilling the less interesting ones to disk. This assumes that the fetching cost is significantly faster than the processing cost.

[Shreya: TODO: rest of work]

9.5 BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data [2]

In this paper, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 100 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 200× faster than Hive), within an error of 2-10%.

Given a query like SELECT AVG(gpa) FROM enroll GROUP BY college, gender, BlinkDB will sample enroll stratified on college, gender and use the sampled data to evaluate queries really fast. The sets of columns used by queries in aggregations is called query column sets, or QCs.

BlinkDB consists of two main modules: (i) Sample Creation and (ii) Sample Selection. We sample creation module creates stratified samples on the most frequently used QCSs to ensure efficient execution for queries on rare values. By stratified, we mean that rare subgroups (e.g., Galena, IL) are over-represented relative to a uniformly random sample. This ensures that we can answer queries about any subgroup, regardless of its representation in the underlying data.

For sample creation, they take a collection of historical QCs and their frequencies and choose a collection of stratified samples with total storage costs below some config threshold. Based on a query's error/response time constraints, the sample selection module dynamically picks a sample on which to run the query. It does so by running the query on multiple smaller sub-samples (which could potentially be stratified across a range of dimensions) to quickly estimate query selectivity and choosing the best sample to satisfy specified response time and error bounds. It uses an Error-Latency Profile heuristic to efficiently choose the sample that will best satisfy the user-specified error or time bounds.

9.5.1 Background

There are 4 approaches to AQP on a spectrum of performance and query flexibility:

1. Predictable Queries: At the most restrictive end of the spectrum, one can assume that all future queries are known in advance, and use data structures specially designed for these queries
2. Predictable Query Predicates: Here, we assume that we know the predicates used in queries, but don't know the queries themselves. The frequencies of group and filter predicates don't change over time.
3. Predictable QCSs: We assume that we know the columns used in the predicates of future queries, but we don't know the queries and we don't know the values being used in the predicates.
4. Unpredictable Queries: If we don't assume anything about queries, then we have to use something online aggregation, which can degenerate into a full table scan

Surprisingly, over 90% of queries are covered by 10% and 20% of unique QCSs in the traces from Conviva and Facebook respectively.

Question 9.4. Discuss the observation BlinkDB made when analyzing historical OLAP queries from Facebook/Conviva, and how that led to the BlinkDB contribution.

9.5.2 System Overview

BlinkDB extends the Apache Hive framework by adding two major components to it: (1) an online sampling module that creates and maintains samples over time, and (2) a run-time sample selection module that creates an Error-Latency Profile (ELP) for queries.

BlinkDB only supports SELECT-FROM-WHERE-GROUPBY-HAVING queries without joins or subqueries. Queries are like:

```
SELECT COUNT(*)
FROM Sessions
WHERE Genre = 'western'
GROUP BY OS
WITHIN 5 SECONDS
```

which will return the most accurate results for each GROUP BY key in 5 seconds, along with a 95% confidence interval for the relative error of each result. Alternatively, one can specify `ERROR WITHIN 10% AT CONFIDENCE 95%`.

9.5.3 Sample Creation

BlinkDB creates a set of samples to accurately and quickly answer queries. First, for each unique value of the group-by clause, we randomly sample it proportional to its representation (overall constructing a stratified sample). We store each stratified sample contiguously in a set of blocks.

Based on the specific query, at runtime we will need to determine the number of rows n for the result (to satisfy error or time bounds).

They employ an optimization algorithm to select a set of QCSs to sample, given a space budget. Thee optimization problem takes three factors into account in determining the sets of columns on which stratified samples should be built: the "sparsity" of the data, workload characteristics, and the storage cost of samples. They try to optimize for the weighted sum of the coverage of the QCSs of the queries (the product of query probability, query coverage, and query sparseness), subject to space constraints. Note that even partial coverage is good.

Question 9.5. Explain what BlinkDB is trying to optimize in creating the samples.

9.5.4 BlinkDB Runtime

Picking a sample involves selecting either the uniform sample or one of the stratified samples (none of which may stratify on exactly the QCS of the current query), and then possibly executing the query on a subset of tuples from the selected sample.

If there is a QCS superset materialized, great—we use that. Otherwise, we use the least selective QCS that is a subset of the query.

An error profile is created for all queries with error constraints. If Q specifies an error (e.g., standard deviation) constraint, the BlinkDB error profile tries to predict the size of the smallest sample that satisfies Q's error constraint. Similarly, a latency profile is created for all queries with response time constraints. To determine what sample size to use (and maybe the QCS), we run the query on various subsample sizes to estimate the error and latency. We use the ELPs to choose the subsample size.

Running a query on a non-uniform sample introduces a certain amount of statistical bias in the final result since different groups are picked at different frequencies. In particular while all the tuples matching a rare subgroup would be included in the sample, more popular subgroups will only have a small fraction of values represented. To correct for this bias, BlinkDB keeps track of the effective sampling rate for each group associated with each sample in a hidden column as part of the sample table schema, and uses this to weight different subgroups to produce an unbiased result.

Question 9.6. How does BlinkDB debias results?

BlinkDB periodically refreshes samples and also periodically refreshes the set of QCS that chooses to sample.

Question 9.7. What's the difference between "predictable queries" and "predictable query predicates"?

10 Special-case Data Models: Streams, Semistructured, Graphs

We cover:

- The CQL Continuous Query Language: Semantic Foundations and Query Execution
- Dataguides: Enabling query formulation and optimization in semistructured databases
- PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

10.1 The CQL Continuous Query Language: Semantic Foundations and Query Execution [5]

Abstract

CQL, a Continuous Query Language, is supported by the STREAM prototype Data Stream Management System at Stanford. CQL is an expressive SQL-based declarative language for registering continuous queries against streams and updatable relations. We begin by presenting an abstract semantics that relies only on "black box" mappings among streams and relations. From these mappings we define a precise and general interpretation for continuous queries. CQL is an instantiation of our abstract semantics using SQL to map from relations to relations, window specifications derived from SQL-99 to map from streams to relations, and three new operators to map from relations to streams. Most of the CQL language is operational in the STREAM system. We present the structure of CQL's query execution plans as well as details of the most important components: operators, inter-operator queues, synopses, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the Linear Road benchmark recently proposed for Data Stream Management Systems. We also curate a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

A database management system (DBMS) allows user to write ad-hoc queries against a static (or slowly changing) database. A data stream management system (DSMS) allows users to register stream queries which continuously run against streamed inputs. You register a set of continuous queries ahead of time and then throw streams of data at it. The DSMS automatically executes your continuous queries as stream data pours in. For example, your stream might consist of sensor readings reporting the speed and location of cars on a highway, and your DSMS stream out toll information for the cars.

This paper presents a (somewhat) formal semantics for an abstract continuous query language and instantiate the abstract query language with a concrete query language CQL which the authors have implemented in their DSMS STREAM.

10.1.1 Introduction to Running Example

We introduce a running example based on a hypothetical road traffic management application introduced in the Linear Road benchmark for data stream management systems. Imagine we have L bi-directional 100-mile long highways that are divided into 100 1-mile long segments. Every car that drives on these highways is fitted with a sensor that reports the speed and position (highway number, x position, direction) of the car every 30 seconds. Our goal is to assign a toll to each car in a congested segment of highway. We say a segment is congested if the average speed of the cars in the segment over the last 5 minutes is less than 40 mph. The toll we apply is $basetoll \times (numvehicles - 150)^2$. Our goal is to take in the sensor reading stream and stream out tolls.

10.1.2 Streams and Relations

Definition 10.1. A **stream** S is a (possibly infinite) bag (multiset) of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and $\tau \in T$ is the timestamp of the element.

There are two classes of streams: base streams, which are the source data streams that arrive at the DSMS, and derived streams, which are intermediate streams produced by operators in a query. They use the term tuple of a stream to denote the data (non-timestamp) portion of a stream element.

Definition 10.2. A **relation** R is a mapping from T to a finite but unbounded bag of tuples belonging to the schema of R. A relation R defines an unordered bag of tuples at any time instant $\tau \in T$, denoted $R(\tau)$.

10.1.3 Abstract Semantics

Operators are in 3 classes:

- stream-to-relation operators: that produce a relation from a stream

- relation-to-relation operators: that produce a relation from one or more other relations
- relation-to-stream operators: that produce a stream from a relation

Diving into the semantics:

- Stream-to-Relation Operators: based on the concept of a sliding window over a stream: a window that at any point of time contains a historical snapshot of a finite portion of the stream. there are time-based, tuple-based, and partitioned.
- Relation-to-Relation Operators: same as in static setting
- Relation-to-Stream Operators: istream (insert stream), dstream (delete stream), and rstream (relation stream). Rstream used mostly with `now` ranges. Istream used mostly with `unbounded` queries. Istream and dstream can be implemented with rstream. By default, unbounded ranges are added, and Istreams are added to monotonic queries.

As syntactic sugar, Istream is added to monotonic queries and an [Range Unbounded] clause is added to streams.

10.1.4 Linear Road in CQL

```
# 1: SegSpeedStr: compute segment speeds for each vehicle
Select vehicleId, speed, xPos/1760 as segNo, dir, hwy
From PosSpeedStr

# 2: ActiveVehicleSegRel: compute last segment for each active vehicle
Select Distinct L.vehicleId, L.segNo, L.dir, L.hwy
From SegSpeedStr [Range 30 Seconds] as A,
SegSpeedStr [Partition by vehicleId Rows 1] as L
Where A.vehicleId = L.vehicleId

# 3: VehicleSegEntryStr: stream of vehicles entering a segment
Select Istream(*) From ActiveVehicleSegRel

# 4: CongestedSegRel: current set of congested segments
Select segNo, dir, hwy
From SegSpeedStr [Range 5 Minutes]
Group By segNo, dir, hwy
Having Avg(speed) < 40
```

```
# 5: SegVolRel: current count of vehicles in each segment
Select segNo, dir, hwy, count(vehicleId) as numVehicles
From ActiveVehicleSegRel
Group By segNo, dir, hwy

# 6: TollStr: final output toll stream
Select Rstream(E.vehicleId, basetoll * (V.numVehicles - 150) * (V.numVehicle
From VehicleSegEntryStr [Now] as E,
CongestedSegRel as C, SegVolRel as V
Where E.segNo = C.segNo and C.segNo = V.segNo and
E.dir = C.dir and C.dir = V.dir and
E.hwy = C.hwy and C.hwy = V.hwy
```

10.1.5 Time Management

Specifically, our continuous semantics is based on time logically advancing within domain T. Conceptually, at time $\tau in T$ all inputs up to τ are processed and the output corresponding to τ (stream elements with timestamp τ or instantaneous relation at τ) is produced.

A heartbeat consists simply of a timestamp $\tau \in T$, and has the semantics that after arrival of the heartbeat the system will receive no future stream elements with timestamp $\leq tau$. There are various ways by which heartbeats may be generated. Here are three examples:

1. If tuples are timestamped by a centralized DSMS, then the DSMS can generate the hearbeats.
2. If all inputs sources deliver tuples in timestamp order, then the input sources can send heartbeats and the minimum of all heartbeats is the true heartbeat.
3. If all input sources have access to a global clock and there is a bounded message delivery delay, then heartbeats can be inferred.

10.1.6 Implementation

CQL is implemented in STREAM. STREAM represents both streams and relations as streams in which each timestamped tuple is also annotated as an insertion or a deletion. Query plans are graphs in which ertices are operators and edges are queues (in memory). Operators read streams and relations (in their unified format) from their input queues and output a stream or relation on their output queue.

Every operator has a corresponding synopsis (in memory) in which it can maintain its state. For example, a sliding window join may create a couple of hash tables. Tuple are not copied when possible. Instead, they are stored in synopses, and tuple references are passed around. STREAM (a DSMS that implements CQL) has physical operators for all of the CQL operators and a couple of lower-level system operators.

Equivalences for optimization: window reduction (istream to rstream) and filter-window commutativity

Question 10.1. Give an example stream query that has ambiguous semantics and explain how CQL makes the semantics clear.

Answer:

```
SELECT *
FROM PosSpeedStr
WHERE speed > 65
```

Question 10.2. What is the relationship between stream processing and materialized views?

Question 10.3. How can we implement Istream and Dstream as an Rstream?

For istream, use select with window now. For dstream, use select with difference.

Question 10.4. Why is a SQL query plan a tree but a CQL query plan a graph?

Synopses & edges/queues are shared between query plans in CQL, so you end up having a graph, or individual query trees stitched together with shared edges.

10.2 Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases [21]

In semistructured databases there is no schema fixed in advance. To provide the benefits of a schema in such environments, we introduce DataGuides: concise and accurate structural summaries of semistructured databases. DataGuides serve as dynamic schemas, generated from the database; they are useful for browsing database structure, formulating queries, storing information such as statistics and sample values, and enabling query optimization. This paper presents the theoretical foundations of DataGuides along with an algorithm for their creation and an overview of incremental maintenance. We provide performance results based on our implementation of DataGuides in the Lore DBMS for semistructured data. We also describe the use of DataGuides in Lore, both in the user interface to enable structure browsing and query formulation, and as a means of guiding the query processor and optimizing query execution.

DataGuides are designed to work with the object exchange model (OEM) which we now describe. In the OEM, data is represented as a directed possible cyclic graph with a designated root such that: each vertex has an object id and each edge is annotated with a label. Leaves are annotated with data. The object exchange model, unlike the relational model, is semistructured. Not all objects have the same set of outgoing labels. In fact, the set of labels is not even predetermined. See Figure 6.

10.2.1 DataGuides

Definition 10.3. A **DataGuide** for an OEM source s is an OEM object d such that every label path of s has exactly one data path instance in d, and every label path of d is a label path of s

Basically, DataGuides have the same distinct paths as the source. So to check if a path exists in the source, one simply needs to check the dataguide. Kind of like NFAs to DFAs.

10.2.2 Existence of Multiple DataGuides

There may exist multiple DataGuides for the same object s (similar to how there are multiple determinizations of an NFA), so there is a question of which to construct. There always exists a minimal DataGuide (as there always exists a minimal DFA), and it seems logical to prefer a minimal DataGuide, however, this is not the case. A minimal DataGuide

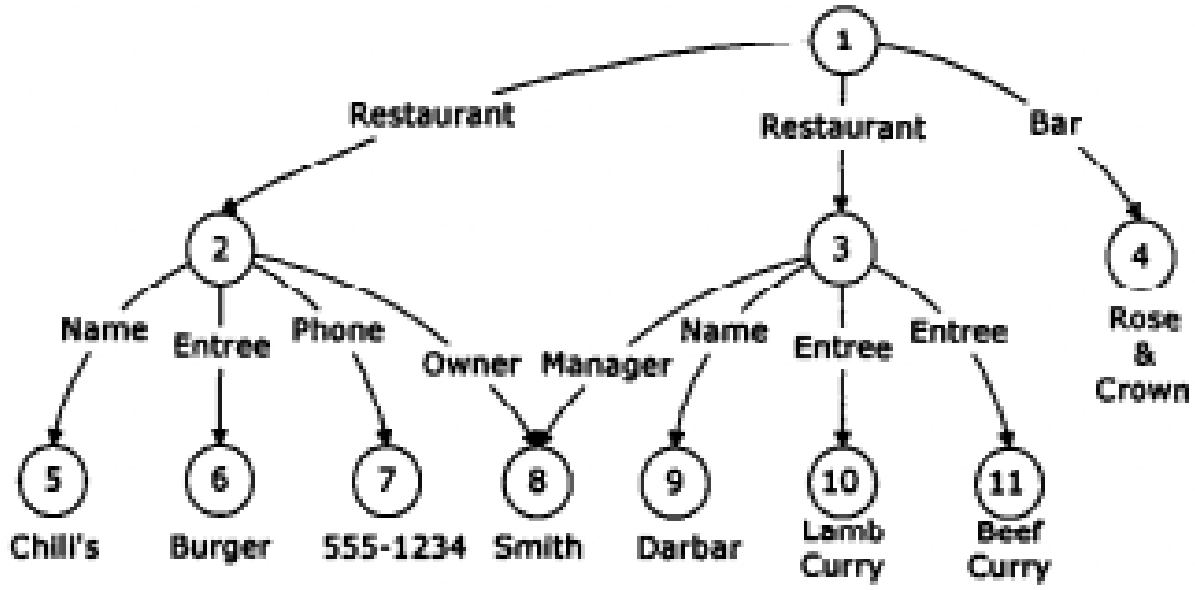

Figure 1. A sample OEM database

Figure 6: Sample OEM DB

is a bad choice for a couple of reasons. For example, a minimal DataGuide can be much harder to maintain than larger DataGuides.

Also, Vanilla DataGuides do not contain any atomic data. However, it is useful to put some data into the DataGuide to describe the source object. For example, a DataGuide could contain example objects from the source object, or it could contain statistics about the distribution of source objects. Formally, we say an annotation of l is some property of a target set $T_s(l)$.

A strong dataguide d of s has a bijection between the target sets (set of object ids reachable from a label path) and the objects in d. It allows us to unambiguously store annotations in the objects of d. To create a strong DataGuide, we simply perform the vanilla determinization procedure. Determinization can take exponential time, but the paper presents some experimental results showing that for typical databases, determinization runs quick enough.

We can do incremental maintenance of the dataguide whenever edges/objects are added to the source. DataGuides expectedly speed up queries for a specific label path. They reduce the cost to time linear in the length of the label path.

A DataGuide acts as an index from a label path to a target set.

10.3 PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs [22]

Large-scale graph-structured computation is central to tasks ranging from targeted advertising to natural language processing and has led to the development of several graph-parallel abstractions including Pregel and GraphLab. However, the natural graphs commonly found in the real-world have highly skewed power-law degree distributions, which challenge the assumptions made by these abstractions, limiting performance and scalability. In this paper, we characterize the challenges of computation on natural graphs in the context of existing graph-parallel abstractions. We then introduce the PowerGraph abstraction which exploits the internal structure of graph programs to address these challenges. Leveraging the PowerGraph abstraction we introduce a new approach to distributed graph placement and representation that exploits the structure of power-law graphs. We provide a detailed analysis and experimental evaluation comparing PowerGraph to two popular graph-parallel systems. Finally, we describe three different implementation strategies for PowerGraph and discuss their relative merits with empirical evaluations on large-scale real-world problems demonstrating order of magnitude gains.

Graph-parallel abstractions rely on each vertex having a small neighborhood to maximize parallelism and effective partitioning to minimize communication. However, graphs derived from real-world phenomena, like social networks and the web, typically have powerlaw degree distributions, which implies that a small subset of the vertices connects to a large fraction of the graph.

10.3.1 Related Work

In graph processing frameworks like Pregel and GraphLab, a graph is distributed across a cluster. To express graph computation, users write vertex programs which are executed in parallel on each of the vertices in the graph. Here, we review Pregel and GraphLab.

Pregel executes in a bulk synchronous parallel fashion in a series of super-steps. In each super step, a vertex program combines messages sent by other vertices using an associative and commutative combiner. The vertex program then updates the vertex's state and sends messages to other vertices. Computation ends when all vertices vote to terminate.

GraphLab executes asynchronously with a shared state abstraction. Data is stored on both vertices (which is accessible to all neighbors of the vertex) and on edges (which is accessible only to the two incident vertices). Vertex programs directly read and write the data stored on neighboring vertices and edges. While vertex programs do not run synchronously, GraphLab ensures the execution is serializable.

Both of these graph frameworks can be abstracted using a single gather-apply-scatter

(GAS) framework in which computation is expressed using a gather (g), apply (a), and scatter (s) function. Data stored on vertex u is denoted D_u, and data stored on edge (u, v) is denoted $D_{(u,v)}$. For a vertex u, accumulators generated by the gather function are summed together. Then, apply generates a new state for the vertex. Finally, edges are updated using scatter: which uses the new value of the central vertex to update the data on adjacent edges.

Pregel and GraphLab do not work well on natural graphs because in these frameworks, the storage, communication, and computation overheads of a vertex are proportional to its degree. The frameworks are able to parallelize different vertices across multiple machines but are unable to parallelize a single vertex program across multiple machines.

Question 10.5. What are drawbacks of Pregel and GraphLab?

10.3.2 PowerGraph Abstraction

From GraphLab, PowerGraph borrows the data-graph and shared-memory view of computation eliminating the need for users to architect the movement of information. From Pregel, PowerGraph borrows the commutative, associative gather concept. PowerGraph vertex programs implement the functions gather, sum, apply, and scatter.

For a vertex u, gather collects data from u's neighbors (specified as none, all, in, or out). The intermediate accumulators generated by gather are accumulated by sum. apply generates a new state. Scatter writes data to u's neighboring edges, can return an optional delta accumulator (more on this later), and can activate other vertexes to run (more on this later too). See the paper for three example PowerGraph programs. To work well on natural graphs, the size of accumulators and the runtime of apply should be constant in the degree of the node.

We can think of gather and sum as map and reduce functions. Then the apply function computes a new vertex, from the final accumulator. Then the scatter phase is invoked on edges adjacent to the new vertex.

Question 10.6. Discuss the gather-apply-scatter framework for parallelizing graph computation.

10.3.3 Other Details

When a neighbor v to vertex u updates its state and activates the execution of u, most of u's neighbors have not changed, so running gather on all of them is largely a waste. The scatter function can optionally return an additional δa which is atomically added to the cached accumulator a_v of the neighboring vertex v using the sum function.

Vertex programs can activate neighboring vertex programs to execute. The order in which vertices execute is controlled by the system. PowerGraph supports two modes of execution: bulk synchronous and asynchronous. In bulk synchronous mode, gather, sum, apply, and scatter are run in lock step across all vertices. In asynchronous mode, vertices are run whenever they are activated, and like GraphLab, PowerGraph ensures serializable execution.

10.3.4 Distributed Graph Placement

In order to execute graph algorithms in parallel, we have to distribute graphs across a cluster of machines. Pregel and GraphLab do so with edge cuts. They divide the vertices of the graph (roughly) evenly across all machines and try to minimize the number of edges which traverse machines. The more cut edges, the more communication and storage overhead. Both platforms randomly sometimes end up assigning vertices randomly across p machines in which case $1 - \frac{1}{p}$ (aka almost all) edges are cut.

PowerGraph executes vertex programs on multiple machines by performing a vertex cut. Edges are assigned roughly evenly to all machines. The greedy algorithm is as follows: for every edge (u, v) where $A(u)$ is the set of machines where u is replicated:

- If $A(u)$ and $A(v)$ intersect, assign the edge to one of the machines in the intersection
- If $A(u)$ and $A(v)$ don't intersect and are nonempty, assign the edge to the machine in $A(u) \cup A(v)$ corresponding to the vertex with the most unassigned edges
- If both $A(u)$ and $A(v)$ are empty, assign the edge to the least loaded machine

Serializability is achieved using a fancy dining philosophers algorithm. Fault tolerance is implemented with snapshot and fancy distributed snapshot algorithms.

11 Data Integration, Provenance and Transformation

We cover:

- Schema Mapping as Query Discovery
- Provenance in Databases: Why, How, and Where
- Wrangler: Interactive Visual Specification of Data Transformation Scripts

11.1 Schema Mapping as Query Discovery [34]

Often people want to migrate data from one schema to another. Typically, this schema migration is done with hand-written one-off scripts which are finicky and hard to optimize. This paper presents an algorithm to semi-automatically infer SQL queries which can perform the migration with a small amount of feedback and interaction from a database administrator. The algorithm is implemented in a system called Clio.

11.1.1 Value Correspondences

Existing work on schema mapping involved set-based schema assertions relating sets of values from the input DB to sets of values in the target DB. This paper proposes value correspondences which instead involve showing how pairs of tuples map to another tuple. This is arguably easier to understand and can lead to automatically finding a query to perform the mapping.

At a high level, a value correspondence is (1) a function mapping a tuple of source tuples to an output tuple and (2) a filter on the source tuples. An example of (1) is prepending a character; an example of (2) is throwing away invalid examples.

Value correspondences are arguably easier to specify than schema assertions. Database administrators do not have to know database-wide schema relations in order to perform a migration. They just have to know how to construct tuples in the output database.

11.1.2 Constructing Schema Mappings

The schema mapping generator should follow two heuristics:

- All values in the source should somehow end up producing a tuple in the target. This favors unions over intersections.

- Values in the source should contribute only once to a tuple in the output. This favors joins over cross products.

11.1.3 Query Discovery Algorithm

[Shreya: TODO: the rest]

11.1.4 Provenance in Databases: Why, How, and Where (Cheney et al.)

Abstract

Different notions of provenance for database queries have been proposed and studied in the past few years. In this article, we detail three main notions of database provenance, some of their applications, and compare and contrast amongst them. Specifically, we review why, how, and where provenance, describe the relationships among these notions of provenance, and describe some of their applications in confidence computation, view maintenance and update, debugging, and annotation propagation.

Data provenance, also known as data lineage, describes the origin and history of data as it is moved, copied, transformed, and queried in a data system.

The lineage of tuple t in the output of evaluating query Q against database instance I is a subset of the tuples in I (known as a witness) that are sufficient for t to appear in the output. Technically speaking, by "witness" we mean a subset of the input database records that is sufficient to ensure that a given output tuple appears in the result of a query.

11.1.5 Why-provenance

Why-provenance is based on the idea of providing information about the witnesses to a query. Recall that a witness is a subset of the database records that is sufficient to ensure that a given record is in the output. There may be a large number of such witnesses because many records are "irrelevant" to the presence of an output record of interest. In fact, the number of witnesses can easily be exponential in the size of the input database. The why-provenance of an output tuple t in the result of a query Q applied to a database D is defined as the witness basis of t according to Q. A minimal witness basis is a witness basis consisting only of minimal witnesses. That is, it won't include two witnesses w and w' where $w \subseteq w'$.

11.1.6 How-Provenance

[Shreya: TODO]

How-provenance uses a provenance semiring to hint at how an tuple was derived. The semiring consists of polynomials over tuple ids. The polynomial $t^2 + t \cdot t'$ hints at two derivations: one which uses t twice and one which uses t and t'.

11.1.7 Where-Provenance

Where-provenance is very similar to why-provenance except that we'll now point at a particular entry (or location) of an output tuple t and ask which input locations it was copied from. For example, the where-provenance of the A entry of tuple t_8 is the A entry of tuple t_3 or t_4.

11.1.8 Eager vs Lazy

Data lineage can be tracked through eager or lazy approaches. In the eager approach, tuples are annotated and their annotations are propagated through the evaluation of a query. The lineage of an output tuple can then be directly determined using its annotations. In the lazy approach, tuples are not annotated. Instead, the lineage of a tuple must be derived by inspecting the query and input database.

11.2 Wrangler: Interactive Visual Specification of Data Transformation Scripts [35]

Abstract

Though data analysis tools continue to improve, analysts still expend an inordinate amount of time and effort manipulating data and assessing data quality issues. Such "data wrangling" regularly involves reformatting data values or layout, correcting erroneous or missing values, and integrating multiple data sources. These transforms are often difficult to specify and difficult to reuse across analysis tasks, teams, and tools. In response, we introduce Wrangler, an interactive system for creating data transformations. Wrangler combines direct manipulation of visualized data with automatic inference of relevant transforms, enabling analysts to iteratively explore the space of applicable operations and preview their effects. Wrangler leverages semantic data types (e.g., geographic locations, dates, classification codes) to aid validation and type conversion. Interactive histories support review, refinement, and annotation of transformation scripts.

User study results show that Wrangler significantly reduces specification time and promotes the use of robust, auditable transforms instead of manual editing.

Data scientists spend an inordinate amount of their time wrangling data. Wrangler makes data wrangling much easier. With Wrangler, analysts specify transformations by building up a sequence of basic transforms. As users select data, Wrangler suggests applicable transforms based on the current context of interaction. Programming-by-demonstration techniques help analysts specify complex criteria such as regular expressions.

The Wrangler transformation language features the following classes of transformations:

- Map: transforms one input data row to zero, one, or multiple output rows
- Lookups and joins (e.g., mapping zip codes to state names for aggregation across states). Currently Wrangler supports two types of joins: equi-joins and approximate joins using string edit distance. These joins are useful for lookups and for correcting typos for known data types.
- Reshape: manipulate table structure and schema with two reshaping operators: fold and unfold. Fold collapses multiple columns to two or more columns containing kv sets; unfold creates new column headers from data values. Kind of like pivoting
- Positional: fill data from surrounding data or move columns up or dawn
- Sorting, aggregation, key generation

11.2.1 Wrangler Interface Design

- Direct manipulation and visual selections
- Automated Transformation Suggestions
- Natural Language Descriptions
- Visual Transformation Previews
- Transformation Histories and Export
- Data quality bar with green (satisfies data type and semantic role), yellow (satisfies data type but not semantic role), red (doesn't satisfy data type), and gray (missing)

11.2.2 Wrangler Inference Engine

The procedure infers transform parameters, generates candidate transforms, and ranks the candidate transforms. Easier transformations are ranked first. It also ensures a diversity of different operations are suggested.

The inference engine relies heavily on a corpus of previous transform frequencies. Since the same exact transform is unlikely to appear before, Wrangler uses a looser notion of transform equivalence. For example, all row selections are considered equivalent, all column selections of the same type are considered equivalent, etc.

12 Systems Support for ML

The papers here are:

- The MADlib analytics library: or MAD skills, the SQL
- HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent
- Scaling Distributed Machine Learning with the Parameter Server

12.1 The MADlib Analytics Library [31]

Abstract

MADlib is a free, open-source library of in-database analytic methods. It provides an evolving suite of SQL-based algorithms for machine learning, data mining and statistics that run at scale within a database engine, with no need for data import/export to other tools. The goal is for MADlib to eventually serve a role for scalable database systems that is similar to the CRAN library for R: a community repository of statistical methods, this time written with scale and parallelism in mind. In this paper we introduce the MADlib project, including the background that led to its beginnings, and the motivation for its opensource nature. We provide an overview of the library's architecture and design patterns, and provide a description of various statistical methods in that context. We include performance and speedup results of a core design pattern from one of those methods over the Greenplum parallel DBMS on a modest-sized test cluster. We then report on two initial efforts at incorporating academic research into MADlib, which is one of the project's goals. MADlib is freely available at http://madlib.net, and the project is open for contributions of both new methods, and ports to additional database platforms.

For data scientists a DBMS is a scalable analytics runtime— one that is conveniently compatible with the database systems widely used for transactions and accounting. With MADlb, for example, you can store labelled training data in a relational database and run logistic regression over it like this:

```
SELECT madlib.logregr_train(
  'patients',                              -- source table
  'patients_logregr',                      -- output table
  'second_attack',                         -- labels
  'ARRAY[1, treatment, trait_anxiety]', -- features
  NULL,                                    -- grouping columns
```

```
  20,                                   -- max number of iterations
  'irls'                                -- optimizer
);
```

MADlib programming is divided into two conceptual types of programming: macro-programming and micro-programming. Macro-programming (orchestration) deals with partitioning matrices across nodes, moving matrix partitions, and operating on matrices in parallel. UDAs are an example, and the library supports scripts as UDAs. Micro-programming deals (data representations, iter loops) with writing efficient code which operates on a single chunk of a matrix on one node.

12.1.1 Macro Programming

UDAs come in 3 pieces:

- Transition function to fold over a set
- Merge function to combine intermediate aggregates
- Final function to apply to the last merged aggregate

Standard user-defined aggregates aren't sufficient to express a lot of machine learning algorithms. A problem is that you can't easily iterate through many passes of the same data. To solve this, the authors store state in between iterations in temporary tables.

12.1.2 Micro-Programming

For UDFs that operate at the row level (perhaps called multiple times per row), the standard practice is to implement them in C or C++. When computing dense matrix operations, these functions would make native calls to an open-source library like LAPACK or Eigen.

Question 12.1. What is the difference between macro and micro programming?

Question 12.2. What are the 3 building blocks of UDAs?

Question 12.3. Why is it hard to use SQL UDAs for ML algorithms?

12.2 HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent [46]

Abstract

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve stateof-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performancedestroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

If each step of a stochastic gradient descent only updates a small part of the weight vector, we say it is sparse. Hogwild! is a lock-free algorithm that implements sparse stochastic gradient descent on a multi-core machine.

12.2.1 Sparse Separable Cost Functions

Definition 12.1. Hogwild! performs stochastic gradient descent on cost functions $f : \mathbb{R}^n \to \mathbb{R}$ of the following form:

$$f(x) = \sum_{e \in E} f_e(x_e)$$

where $e \subseteq \{1, \ldots, n\}$ and x_e are the elements of x indexed by e. A cost function is **sparse** if $|E|$ and n are large, but $|e|$ is small for all e.

Algorithm 1 HOGWILD! update for individual processors

loop
  Sample e uniformly at random from E
  Read current state x_e and evaluate $G_e(x_e)$
  for $v \in e$ **do** $x_v \leftarrow x_v - \gamma G_{ev}(x_e)$
end loop

Figure 7: Hogwild

Question 12.4. What is a sparse cost function?

12.2.2 The HOGWILD! Algorithm

The algorithm is in Figure 7.

The Hogwild! algorithm runs on a multicore machine in which each core has access to E and x (i.e. the weight vector). We assume that scalar addition is atomic (componentwise addition operation is atomic), but vector addition is not.

12.2.3 Fast Rates for Lock-Free Parallelism

Convergence rates can be increased by periodically decreasing the step size γ. This requires periodically synchronizing the cores.

12.3 Scaling Distributed Machine Learning with the Parameter Server [38]

Abstract

We propose a parameter server framework for distributed machine learning problems. Both data and workloads are distributed over worker nodes, while the server nodes maintain globally shared parameters, represented as dense or sparse vectors and matrices. The framework manages asynchronous data communication between nodes, and supports flexible consistency models, elastic scalability, and continuous fault tolerance. To demonstrate the scalability of the proposed framework, we show experimental results

on petabytes of real data with billions of examples and parameters on problems ranging from Sparse Logistic Regression to Latent Dirichlet Allocation and Distributed Sketching.

A parameter server is more or less a distributed key-value store optimized for training machine learning models. For example, imagine we're learning a weight vector $w = (w_1, w_2, w_3)$ using logistic regression. We can distribute w across two shards of the parameter server where one shard stores $(1, w_1)$ and the other stores $(2, w_2)$ and $(3, w_3)$. Worker nodes can then read parts of the weight vector, perform some computation, and write back parts of the weight vector. Communication overhead and fault tolerance is of utmost importance to handle

12.3.1 Architecture

A parameter server consists of a bunch of servers that store weights and a bunch of workers that perform computations with the weights (e.g. compute gradients). Servers are organized into a server group managed by a server manager. Workers are organized into multiple worker groups, and each worker group is managed by a task scheduler. The server manager manages which data is assigned to which server. The task scheduler assigns tasks to workers and monitors progress.

Parameters are stores as key-value pairs. For example, a weight vector $w \in \mathbb{R}^d$ can be stored as a set of pairs (i, w_i) for $1 \leq i \leq d$. To store sparse vectors more efficiently, only non-zero entries of w must be explicitly stored. If a pair (i, w_i) is missing, the parameter server assumes $w_i = 0$.

Data is sent between nodes using push and pull operations. In addition to pushing and pulling entries of w, workers can also register user-defined functions to run at a server. For example, a server can compute the gradient of a regularization term. By default, the parameter server runs tasks asynchronously. That is, if a worker issues a pull or push request, it does not block. However, the parameter server also allows workers to explicitly mark dependencies between different requests which forces them to serialize.

There are 3 levels of consistency:

- Sequential: total serialization
- Eventual: no blocking, run requests whenever you'd like
- Bounded delay: make sure no tasks have $> \tau$ delay, all tasks are processed

Question 12.5. What are the 3 levels of consistency?

12.3.2 Implementation

The servers store the parameters (key-value pairs) using consistent hashing. The parameter server uses chain replication to replicate data. Each node forms a chain with the k previous nodes in the hashing ring. Workers send updates to the master which is chain replicated to the next k servers.

The parameter server tags each range of key-value pairs with a vector clock.

References

[1] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 67–78. IEEE, 2000.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42, 2013.

[3] Rakesh Agrawal, Michael J Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654, 1987.

[4] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[6] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[8] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[9] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[10] Eric A Brewer. Combining systems and databases: A search engine retrospective.

[11] Sergey Brin and Lawrence Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 56(18):3825–3833, 2012.

[12] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment*, 1(1):538–549, 2008.

[13] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[14] Rada Chirkova, Jun Yang, et al. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[18] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[19] David J DeWitt, Shahram Ghandeharizadeh, Donovan A Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The gamma database machine project. 1990.

[20] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[21] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. Technical report, 1997.

[22] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.

[23] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. *ACM SIGMOD Record*, 19(2):102–111, 1990.

[24] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[25] Goetz Graefe. The five-minute rule 20 years later (and how flash memory changes the rules). *Communications of the ACM*, 52(7):48–59, 2009.

[26] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*, pages 209–218. IEEE, 1993.

[27] R Lorie J Gray, GF Putzolu, and IL Traiger. Granularity of locks and degrees of consistency. *Modeling in Data Base Management Systems, GM Nijssen ed., North Holland Pub*, 1976.

[28] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. Datalog and recursive query processing. *Foundations and Trends® in Databases*, 5(2):105–195, 2013.

[29] Ashish Gupta and Inderpal Singh Mumick. On the computation of multidimensional aggregates. 1999.

[30] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing data cubes efficiently. *Acm Sigmod Record*, 25(2):205–216, 1996.

[31] Joe Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. The madlib analytics library or mad skills, the sql. *arXiv preprint arXiv:1208.4165*, 2012.

[32] Joseph M Hellerstein, Ron Avnur, and Vijayshankar Raman. Informix under control: Online query processing. *Data Mining and Knowledge Discovery*, 4(4):281–314, 2000.

[33] Joseph M Hellerstein, Michael Stonebraker, James Hamilton, et al. Architecture of a database system. *Foundations and Trends® in Databases*, 1(2):141–259, 2007.

[34] Mauricio A Hernandez, Renée J Miller, and Laura M Haas. Clio: A semi-automatic tool for schema mapping. *ACM Sigmod Record*, 30(2):607, 2001.

[35] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the sigchi conference on human factors in computing systems*, pages 3363–3372, 2011.

[36] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment*, 7(10):853–864, 2014.

[37] Philip L Lehman and S Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[38] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014.

[39] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. {TAG}: A tiny {AGgregation} service for {Ad-Hoc} sensor networks. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, 2002.

[40] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[41] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[42] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[43] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49, 1997.

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

[45] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

[46] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24, 2011.

[47] P Griffiths Selinger. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[48] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1907–1922, New York, NY, USA, 2016. Association for Computing Machinery.

[49] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[50] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[51] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5):172–182, 1995.

[52] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[53] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.