CS 165 Main Memory Column Store Database

Steven Kekacs

December 11, 2017

Contents

Introduction

Over the last 20 years we have expierienced a rapid growth in technology which has driven an exponential increase in the amount of data that exists to be explored. As the size of data continues to get bigger and bigger so does the importance of having optimized data storage systems. The ability to store and quickly acess vast amounts of information is critical to the success of companies and the discovery of new insights in all fields.

With more thought going into data storage design, many new ideas are being explored to improve both the speed and reliability of storage and access. One of these big ideas is using a column store design rather than the traditional row store design. All of the traditional concepts stay the same, but are implemented in ways to optimize data access and movement for a different storage system. The goal of this project is to design and implement a main memory optimized column store database that includes the key concepts of any modern database.

The main memory column store database I built implements all of the basic operations any database would support, including scans, fetches, updates and aggregates, but also explores more advanced topics such as batched queries for improved scan times, clustered and unclustered sorted and b-plus tree indexes, and nested-loop and more complex hash joins. In building the necessary components for each aspect of the database design choices were made to attempt to minimize data movement and maximize processing power to provid both fast and reliable operations.

Basic Column Store

In Milestone 1 of this project I implemented the core functionality of a column-store database for integer data, with the ability to run single-table queries. The goal of this milestone was to have the ability to create a database with tables and columns, insert data into created tables, and query a table and its columns using select, fetch, print and several aggregate operators, and store the results in intermediary variables that can be used in other operations. Finally, all contents of a created database should be persistent on shutdown.

Design

In order to implement this functionality I had to tackle several different problems, including designing the basic database structure, the database catalog and variable pool to provide quick lookup to db objects and client variables, the system of database operators, and database persistence.

Database Structure

A database is created using three main structures:

- **Db** - contains all database metadata and pointers to tables belonging to db
- **Table** - contains all table metadata and pointers to columns belonging to table
- **Column** - the core struct of a database, contains column metadata and array of data

A created database can have as many tables as needed, but each table has a predetermined number of columns, which is passed as an argument on creation. The data stored in each column is dynamically resized in an amortized way, so as many rows as can fit in main-memory can be insterted.

Database Catalog and Variable Pool

I needed a way to quickly acess database objects to be used in queries, then also a way to store intermediary results from queries in a variable pool that can then be quickly access for use in later queries. Firstly, to store itermediary results I created a Result structure, that is similar to a Column but instead stores an arbitrary data type array. Secondly, to provide fast storage and access of database and result objects, I created a simple hash table that stores Tables, Columns and Results. One, called **db_catalog**, stores all objects pertaining to the created database. The second, called a **client_lookup_table**, is specific to each client connected to the server, and stores result objects.

Database Persistance

While each **client_lookup_table**'s memory can be released on shutdown, the contens of the **db_catalog** must be persistent, i.e. saved to disk on shutdown. In order to ensure persistence of the database I have a system that simply dumps the database to a binary file, table by table, and column by column. Any new features that were later added, such as indexes, were incorporated into this shutdown function so all database relation information was durable. Then, when running a new server, the system first checks if there is a dumped file to load contents from, and if so recreate all objects in main memory.

Database Operators

Finally, we needed a system to actually perform database operations, including the create operators, but also insert, select, fetch and aggregate queries. The created system simply parses a command, collects and verifies the relevant information for the specified query, then creates a DbOperator object that contains this information. If the DbOperator is valid, it is then executed by the appropriate function for its type.

Fast Scans: Scan Sharing & Multi-Cores

While getting correct results is a goal of a database, one accomplished in the first milestone, another goal of any database is to provide fast results. In Milestone 2 of this project I explored several methods to make scanning data fast. This involved minimizing data movement by sharing data between multiple scans and increasing cpu usage by utilizing multiple cores to execute scans.

Design

To enable shared scans I created a way to batch operators together, simply by signaling the start and end of your group of operators to be batched using the batch_queries() and batch_execute() commands, respectively.

Shared Scan

In order to optimize batched queries, selects on the same data should scan together rather than each separately pulling the same data into memory and scanning. This will minimize data movement and increase scan speeds. To implement this I simply modified my scan operator to take multiple queries, and when I iterate through the data to check if each value qualifies, I check each query on each value. This minimizes data movement by processing all of a page of the base data at a time for all queries, rather than pulling in page by page for one query, then doing it again for the next and so on.

To measure the speed-up gained by implementing shared scans, I ran 100 scans of various selectivities on a column of 1 million integers on an increasing number of scans, measuring the time it took for each scan using all individual scans and then shared scans. This expirement was run on an amazon ec2-instance running a linux os with 4-cores and 30.5 GiB of memory.

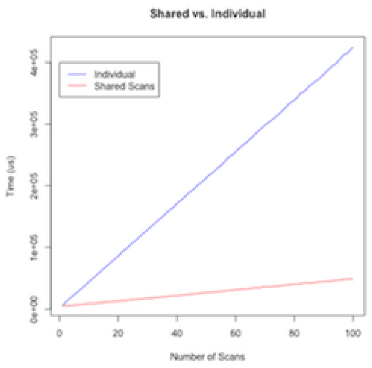

Figure 1: Graph depicting execution time of shared scans vs. individual scans

As shown by the performance graph, sharing scans to minimize data movement provides a drastic improvement in execution time as the number of scans increases.

MultiCore

The other method of improving scan times I explored was utilizing multiple cores to process the data rather than having everything run on a single core. I expiremented with two ways of utilizing multiple cores in shared scans. The first way was to divide up all shared scans into smaller groups of scans that could be ran on seperate cores. That way, data movement is still minimized by sharing scans, but more processing power is used to execute the groups of shared scans in parellel. To figure out the best size of grouped shared scans I ran an expirement using different shared scan chunk sizes (again on linux instance with 4 cores and 30.5 GiB of memory).

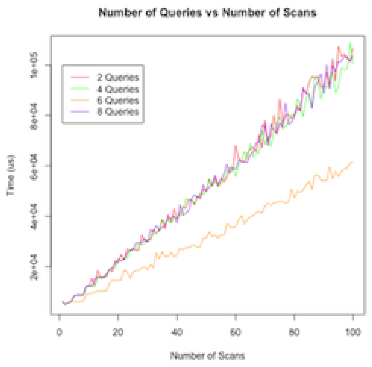

Figure 2: Graph of execution time of paralellized shared scans with different number of queries per thread

The expirement indicates that dividing the queries into sets of 6 shared scans per thread provided the best performance.

The second way of utilizing multiple cores for shared scans I expiremented with was splitting the data being scanned among multiple cores, all executing a shared scans with all batched queries, then reconstructing the results from each thread. Data movement is minimized within each core, as multiple scans are being shared, and processing power is again maximixed by running scans on seperate pieces of the data in parallel. To test the optimal number of threads to chunk the data I ran an expirement using varying numbers of threads to execute the multicore shared scans.

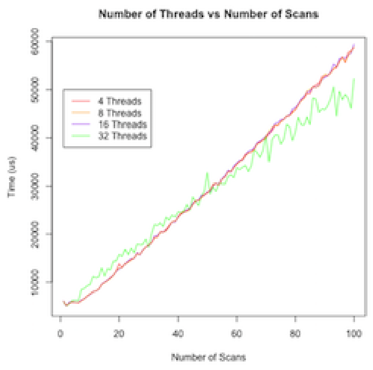

Figure 3: Graph of execution time of paralellized shared scans with varying number of threads

The results show that there is no measurable different between the performances of using 4, 8 or 16 threads, but that using 32 threads was slower for < 50 scans, but better for > 50 scans. Thus, the system uses 4 threads for less than 50 queries and 32 when there are 50 or more queries being batched together.

Lastly, to decide which technique of parallelization provided better performance I compared the results of parallelizing groups of queries and paralellizing chunks of data.

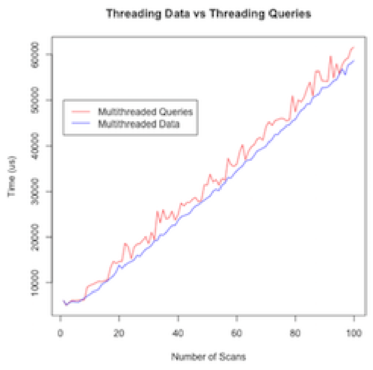

Figure 4: Graph of execution time of paralellized groups of queries vs. paralellized chunks of data

The graph indicates that although the performance difference is minimal, paralellizing shared scans by splitting the data among threads and executing all shared scans on each performs consistantly better.

Lastly, the following graph measures the speedup gained by using multiple cores for shared scans rather than a single core.

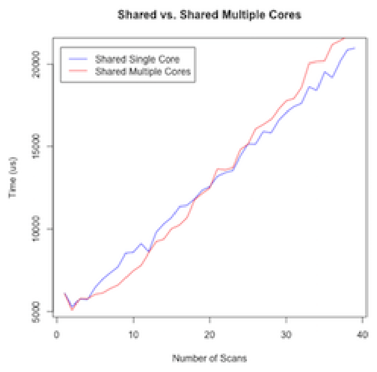

Figure 5: Graph of execution time of multicore shared scans vs. single core shared scans

The performace graph shows that for less than 20 scans, multicore shared scans outperforms the single core shared scans, but for more than 20 scans using just one core seems to perform better. Our system thus executes the appropriate design for the given number of scans.

Performance Analysis

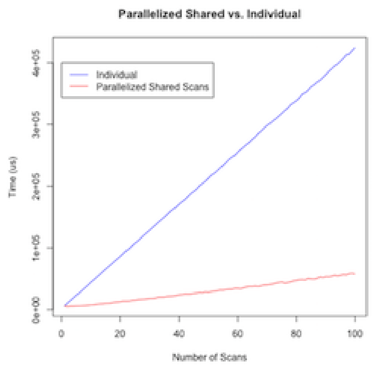

Figure 6: Graph of execution time of individual scans vs. parallelized shared scans

The results show that there is a significant performance gain from implementing optimized, paralellized shared scans. The minimization of data movement and the increased use of processing power significantly reduce the execution times for multiple queries on the same data.

Indexing

In Milestone 3 I added indexing support in order to further improve select operators' execution time. This involded implmenting memory optimized B+ Tree indexes and sorted columns in both a clustered and unclustered form.

Design

In order to support the creation of indexes I added a new create operator type for indexes, and options to specify which type of index it is (btree vs sorted, clustered vs unclustered). For clustered indexes, the order of the column is propogated to all other columns in the table, and for unclustered indexes a copy of the base data was stored in the specified index structure.

Sorted Index

The first type of index I implemented was simply a sorted index. For clustered sorted indexes, this involved simply sorting the base data and inserting at the correct position on each insert, and propogating this position to all other columns. For unclustered sorted indexes, I simply create a copy of the base data which is a sorted array of the values and their corresponding positions in the base data.

B+ Tree Index

The second type of index I implemented was a b+ tree index. My B+ tree is designed to minimize data movement and random accesses. The structure of my B+ tree nodes is as follows:

```c
typedef struct BPTreeInternalNode {
    struct BPTreeNode* pointers[FANOUT];    // array of pointers to other nodes
    int vals[FANOUT - 1];                   // array of keys
} BPTreeInternalNode;


typedef struct BPTreeLeafNode {
    int vals[LEAF_SIZE];          // array of values
    int positions[LEAF_SIZE];     // array of corresponding positions in base data

    struct BPTreeNode* next;      // pointer to next leaf
    struct BPTreeNode* prev;      // pointer to previous leaf
} BPTreeLeafNode;


typedef union BPTreeNodeType {
    BPTreeInternalNode internal_node;
    BPTreeLeafNode leaf_node;
} BPTreeNodeType;

struct BPTreeNode {
    int is_leaf;                  // bool for leaf
    int num_vals;                 // number of vals stored
    BPTreeNodeType type;          // leaf or internal

    struct BPTreeNode* parent;    // pointer to parent node
};
```

I used different designs for internal nodes vs leaf nodes in order to maximize the number of values and positions stored in each leaf as the leaves are storing position values rather than pointers. In order to minimize data movement and random data accesses I made sure each node fit on an some multiple of the cache page size, which is 4096 bytes. To figure out the optimal number of pages for each node I ran an expirement that inserted various numbers of rows into a table and ran 100 selects of varying selectivity for node sizes of 1 page, 4 pages, and 8 pages.

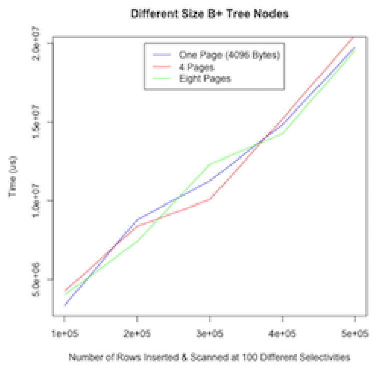

Figure 7: Graph of execution time of different sized B+ Tree Nodes

The results show no significant difference in performance from different sized nodes, so the system sticks to nodes of one page.

Performance Analysis

To compare the speeds of each type of index I ran an expirement that executes 100 selects of increasing selectivity for a column with a sorted unclustered index, a b+ tree unclustered index, and no index.

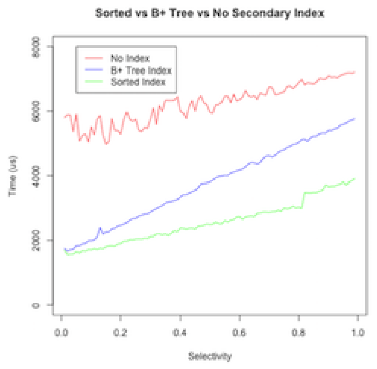

Figure 8: Graph of execution time of indexes

Clearly, both the sorted and B+ tree provide significant speedups compared to the select with no index. The B+ tree index and the sorted index perform similarly for low selectivities, but the sorted index outperforms the B+ tree index as selectivity increases.

Joins

In Milestone 4 I added an integral operation of any database, joins. I implemented two types of join algorithms, a nested-loop join and grace hash join, both built using cache-conscious techniques.

Design

Nested-Loop Join

The nested-loop join was implemented using four nested loops. The outermost loop loads a page of the bigger column being joined, the second outermost loop loads a page of the smaller callem being joined, and the inner two nested loops iterate over each combination values in the pages of the bigger and smaller columns, adding resulting indices where the smaller column value equals the bigger column value to a results array. This technique minimizes data movement by pulling each column a page at a time into the cache, processing that data, then moving to the next page.

Grace Hash Join

The cache conscious hash join I implemented was a grace hash join, where each column of data was partitioned into smaller pieces, on which hash joins were performed. The results from each partition were then reconstructed into a final results array.

In order to minimize cache misses and the cost of resizing a hash table with fixed size, I implemented extindible hashing. This involved keeping a directory of the number of bits of each hashed value to use to lookup the correct bin, then doubling the directory and splitting the appropriate bucket when full.

In order to figure out the optimal number of partitions to use in the hash join I tested three different values, 64 partitions, 128 partitions and 256 partitions, and compared their performance on joins of varying sizes.

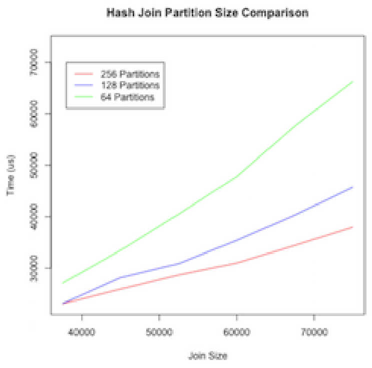

Figure 9: Graph of execution time of grace hash join using different numbers of partitions

The results indicate that using 256 partitions is optimal for all join sizes.

Performance Analysis

The following graph measures the performances of the two types of joins I implemented on different join sizes.

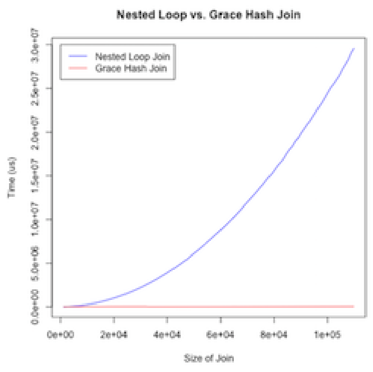

Figure 10: Graph of execution time of nested-loop join vs. grace hash join

The speeds of the nested-loop join and grace hash join on small join sizes is similar, but for larger joins the results show that the grace hash join significantly outperforms the nested-loop join.

Updates

In the final milestone I added update support, allowing read queries to interleave updates (inserts, deletes and actual updates) and maintaining the correctness of any indexes that exist on the base data.

Design

I first implemented a delete operator, that given a table name and list of row ids, removes the given row ids from the given table. This operator simply iterates over the positions to delete, and removes them from the base data and any indexes that exist, making sure to shift data over appropriately and update the indexes.

To implement updates, I utilized the existing insert and delete operator to model an update as a delete followed by an update. Given a column and positions to update, this operator compiles the new row values for the whole table, which is just the existing ones for the columns not being updated, then the new value for the column being updated, deletes the positions from the table, then inserts the new values. This ensures that all updates persist and any existing indexes are correct.