Project Proposal

James Bilous

Jorge Guevara

Brandon Leventhal

Yuliya Levitskaya

Casey Sheehan

CSC 560: Selected Topics in Database Management Systems

Dr. Alex Dekhtyar

October 14, 2014

Abstract

With the increase of data in the world, businesses are looking for ways to store it efficiently. Some of them are finding that they do not need all the power that a relational database provides and are looking for other "lightweight" options. In this document we aim to decompose the design of such a system. We call it MaybeSQL.

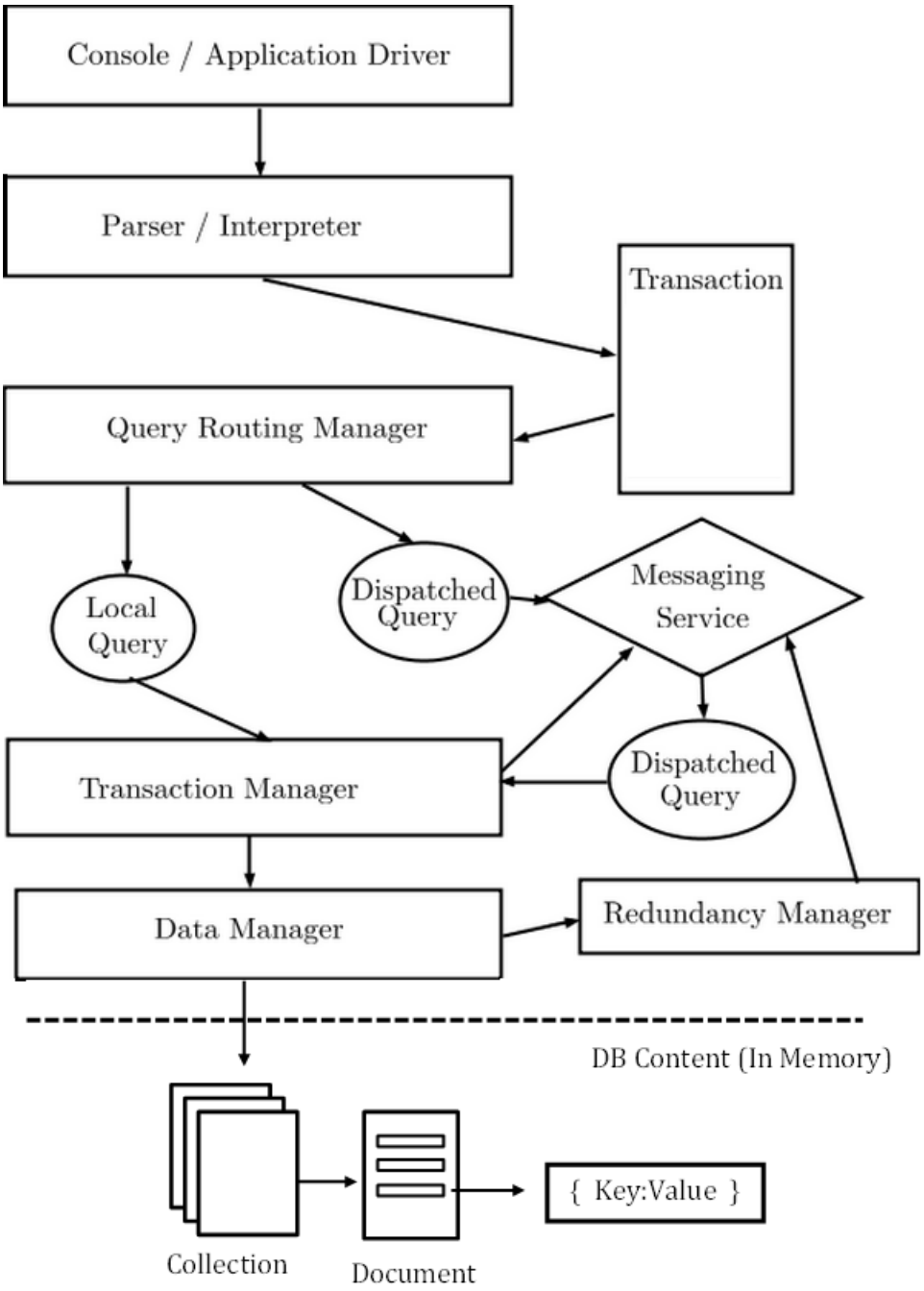
Contents

1 Description

We will be implementing a document database similar to MongoDB. It will be in-memory, so the interaction with the file system will be minimal; mostly to save snapshots of the current state of the database. We will not be supporting transactions and our query language will support put(), get() by key, delete() and count(). We do plan to implement all the other traditional parts of a database, however.

2 Proposed Architecture

2.1 Architecture Diagram



2.2 Components

2.2.1 Parser

The parser is responsible for accepting the input language and tokenizing it to form queries. This is also where syntax errors are reported back to the user. After syntax errors are reported, the parser detects and rejects semantic errors such as nonexistent tables and columns or type mismatches.

We will support three commands as part of our accepted syntax: get(), put() and find(). Get will retreive a document based on a primary key while put places an object into the store and find retrieves a collection of documents based on a non-primary key.

2.2.2 Compiler

The compiler is responsible for taking the tokenized queries and formating them so it can be consumed by the Query Master. By the time information gets to the compiler, it is tokenized and checked for errors.

2.2.3 Log

Log is responsible for keeping track of each get and put. The log is a stable storage that these requests are written to before modifying the database. It is made up of a sequence of records, which maintains the records of actions performed by a transation. It is important that the logs are written prior to the database modification in case of a crash. We have excluded logging from our implementation in favor of duplication of content across neighboring nodes.

2.2.4 Node Hierarchy

There are a total of three nodes. Each node runs a full, independent stack of the DBMS. Queries are routed to other nodes via a message passing depending on where information is stored and where it should end up.

2.2.5 Optimizer

The optimizer is responsible for determining the most efficient way to execute a given query. Query results are generated by accessing relevant database data and manipulating it in such a way that results in the requested information. In most cases, the needed data can be collected from the database by accessing it in different ways, orders, and different data-structures. Each way, typically, requires different processing time. The role of the optimizer is to find that way to process a given query in minimum time. We will simplify our design by ignoring query optimization.

2.2.6 Redundancy Manager

The redundancy manager ensures durability by facilitating duplication of content across nodes. This functionality will mimic MongoDB's "sharding" techniques which allows duplication across several nodes. We will aim for one duplication.

2.2.7 Buffer Manager

The buffer manager is responsible for fetching data from disk storage into main memory and deciding what data to cache in main memory. Buffer manager responds to the request for main memory access to disk blocks. It allocates buffers in virtual memory and lets the operating system decide which buffers should be in main memory and which should be in disk. If requests exceeds available space, the buffer manager has to select a buffer to empty by returning its contents to disk. The buffer manager also has the option of allocating more buffers if they can fit in main memory.

2.2.8 Buffers

Since we will be developing an in-memory database, the buffers will essentially "be" the database. At some point, during the execution of the system, we will have to store a snapshot of the data in the filesystem. At that point, we will have to flush the buffers.

2.2.9 Data Representaton

Pursuant to the data representation model of MongoDB, all data in our database will be stored in BSON documents. These documents are loosely defined in structure, though document size will be necessarily limited to 16MB due to the constraints of BSON.

These documents will then be arranged into groupings, called collections, which will further loosely encapsulate the data. Collections may be thought of as an array of BSON documents, with one document reserved for the storage of metadata. In practice, these documents will share a similar structure, though such will not be strictly enforced by the collection. This allows for the essential elements–that is, those which exist in every document–to be mapped to as a specific object. Documents within a collection may thus be queried on through reference to the a subset of the elements contained within these objects.

2.2.10 Internal Communication

Internal communication will be provided via message passing via MPI or a similar library. Communications will be sent and received asynchronously by the Messaging Module to ensure that there is little to no blocking.

2.2.11 Transaction Manager

The transaction manager is responsible for handling transactions from query routing managers and scheduling them in a way that maintains consistency and isolation across commits. A transaction is comprised of multiple operations that must be performed atomically via communication with the data manager. The transaction manager will also forward transactions to the redundancy manager in order to mirror data across nodes and maintain durability. In order to simplify our implementation, we will assume that transactions contain only a single get or put operation.

3 Implementation Notes

We have decided to implement our database in Java due to everyone being comfortable programming in it and the availability of Open MPI, an open-source library that we will be utilizing. MPI stands for Message Passing Interface and it is typically used for parallel or distributed computing. Using the MPI API we are aiming to get at least three servers running in our database cluster. Each node in the cluser will run the entire DBMS stack and use MPI to pass queries.