Advanced Software Engineering

Enhancement Proposal

HBase

Dec 4, 2018

Daniel McVicar	213027479
Yahya Ismail	213235403
Daniel Fortunato	216796443
Adham El Shafie	212951018
Rafay Sheikh	213033451

Abstract

This report focuses on the concept of enhancing HBase specifically but can also relate to other distributed databases or Bigtables. Here, a feature is proposed that should benefit HBase and many, if not all, of its users. This proposed enhancement is not under development or already implemented as a feature within HBase.

This report then goes into detail on this enhancement, outlining the benefits of it and any drawbacks it may have, along with supported research references, diagrams and figures. Also provided within this report are 2 approaches to implementing this feature that have advantages and disadvantages, along with a final decision on which one would be the most beneficial to introduce into HBase.

Effects of this enhancement on the current version of HBase is discussed, relating specifically to aspects such as maintainability, performance, evolvability and testability. Impacts on the current architecture is also explored with regards to both high level conceptual architecture and low level.

Risks and limitations as well as compatibility testing of this enhancement are both covered, after the enhancement is explained in detail, as well as the features effect on concurrency as it is in HBase.

Introduction

An enhancement is “an increase or improvement in quality, value, or extent” [1] Simply reading through this statement, anything that improves on the current version of a product all while maintaining all its other features is considered an ‘enhancement’. Whether it be improving the speed, quality, quantity or even exposure; they are all enhancements. This doesn’t change when it comes to applications at all. A program or application aims towards completing a certain task given to it by its

developer. Once the original version of the product is created, newer updates to it that maintain its original functionality but improve on any of its qualities are considered enhancements.

Enhancing a program is not something that is “completed”, per se; there can always be improvements or newly added features. Even a program such as HBase can be improved on. HBase is not a ‘perfect’ system for what it does - otherwise there would be no competitors - but does what it should do *very* well. The idea to enhancing an application like this is in researching its disadvantages and what it can’t do, and creating an addition to make up for this drawback. This is what we will be focusing on in this report.

Proposed Feature

Our proposed feature is to implement join operation capabilities in HBase. This is an enhancement that will provide a simple and time saving method to computing a join in HBase. Of course, a “Join” command will be added to list of HBase shell commands list so clients can use this feature when performing query operations. We will illustrate two approach that we can take to implement this feature and after some rational reasoning we will determine which approach is the best way to go.

A join operation is basically combining data (usually 2 or more tables) with matching columns (which could be based on condition) such that they create a relationship between the data. Data is returned in the form of a table that satisfies the join condition. There are many types of join operation that a client may perform. The 4 common and obvious types of joins between two tables: Inner join, Outer join, Left join, Right join which is illustrated in Figure 1. No matter what type of join is used the way the systems work internally will be the same. The 4 joins listed here are just to show that joins can be of different types and why users perform these queries to extract different sets of data. Calling each will result in the same program traceback.

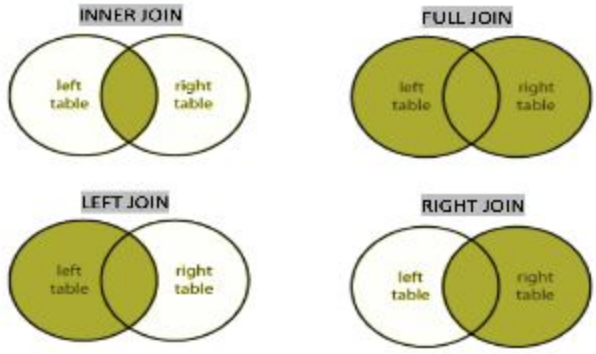

Figure 1 -The usual 4 types of join that can be used

Motivation

Our motivation comes from the few points we research and realized would make this feature useable to wide range of clients. Join operations (or any cross computation between two tables) in any distributed database including SQL distributed databases is actually difficult and use avoided because of the performance cost. We are aware a join operation in HBase will not be 100% efficient or reduce performance cost but users working with not too big of data can still benefit from this. There are case in real life where the result is needed but companies are okay with the time and performance cost required to perform the operation. So, this motivated us because for the user/client this will be an option available for them to use and we will see later on one of the approaches we use does not affect the architecture design/pattern of HBase and does not affect the performance of standard HBase use.

HBase uses a Query-First design, this is not a schema but more like a design pattern that a HBase administration or client should adapt to if they want to get the most out of HBase. Query-First basically means that before using or inputting data in HBase – the queries (type of queries) should be identified first. This is important because in HBase we can use this to store data together as well as read them together. If a client knows he will be performing join operation on a table of set of data, then they can come up with the queries beforehand and use that information to best store data in the regions in different region servers for faster access for read and write operation.

Also, HBase stores its data in denormalized form. A RDMS stores them in normalized form. What we found out that is join operations work better with denormalized data then they do with normalized data. It works better because it is less costly because denormalized data gives you data independence as similar data is grouped together. Using that with above motivation of using query first to design queries we can choose the best location in the regions to store data we know that join operations will be called upon.

HBase - Current State (Problems)

A join operation function is needed because at the moment HBase does not have any built-in method to perform this through queries or any other means. Join operation is a very common operation that users call upon when working with their data in HBase. Yes, it's true that HBase and all other NoSQL, column based, non-structured databases are not built to have these operations supported - or to be exact be

efficient and optimized in performing any cross-reference computation between two tables. Yet, clients and users working with not too large data still find it useful and efficient to perform these operations on data in HBase.

At the moment if a client wants to manipulate their data in HBase with any join or cross computation between two or more tables – they have to use external frameworks and software's to perform these operations on the data. These frameworks and software include use of Apache (Hive, Drill, Phoenix), data has to be imported to these applications and manipulated within before being exported back into HBase, so the client can retrieve it. These external frameworks require extra setup and configurations and are time consuming to perform a simple join operation.

Also, one of the most common methods that users perform to compute a join operation is using Apache MapReduce and even that is not fully optimized to compute such function. There are two methods in MapReduce - a Map side join and a Reduce side join. Both sides require the user to write code on their own in order to make the operation work.

Stakeholders

There are many stakeholders present if this feature is to get implemented in HBase. There are those who will play a part in either the usage (benefit from) of this feature and those who will be vital in the developments and competition this enhancement will bring to an application like HBase.

- **Client**
  - **User (Single):** A user that has HBase setup on a standalone local server or a client who is just starting to understand and use HBase will benefit from this feature. A join operation is very common operation that clients like to perform. They go the extra length to set up external frameworks like Apache Hive, Apache Phoenix, Apache Drill to import their data(tables) and perform these operations and then bring it back into HBase.
  - **User (Small/Medium company/organization):** Clients running HBase to handle not overly large data might find this to be very useful and a time saver. As mentioned before join operations are a very common operation invoked by users and one of the best ways to find a relation between two tables is with a join operation (condition based). A small to medium sized company can really get the most in terms of performance and time costs with the addition of this feature implemented within HBase.

- **Developers**
  - Developers will be a big stakeholder in this feature as they will the ones who are responsible to create, implement, and maintain this feature. They will need to implement it in a way that this does not affect or break the current usage of HBase and offers a very simple and user-friendly method to allow the feature to function. How well the feature is implemented and how much it can be improved over time is on the developers. They can create a patch or external download for this feature so users/clients using older version of HBase can integrate this feature.
- **Competitors**
  - There are many softwares like Apache HBase such as Google Big Table, Cassandra, and Amazon Redshift and although join operation are not meant to be performed on NoSQL type database the truth is clients and user still find it necessary to perform this operations from time to time. This feature might not be 100% efficient or optimized for join operation but by introducing this feature it might be the click that some users need to go for HBase.

Approach 1 - SymLinks

In Operating Systems, there's a concept called symbolic linking in order to link together physically separate files and folders and make them appear to be in the same location. This soft linking is efficient, smart, and inspirational, and it is the basis of SymLinks.[2] SymLinks takes the concept of file system symbolic linking and applies it in a database sense in order to create a non-intrusive method of implementing a Join operation.

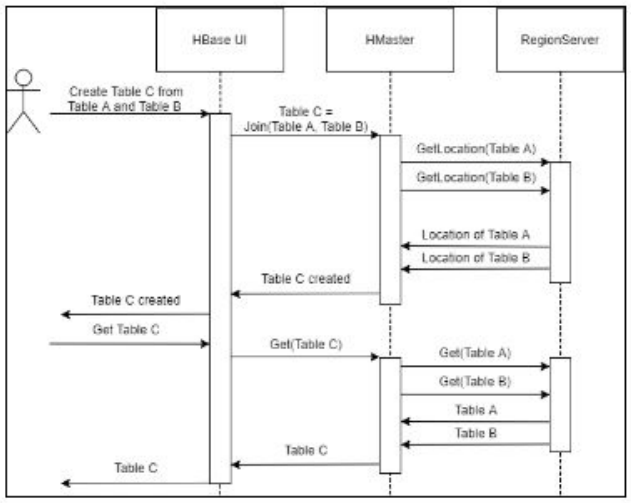

Figure 2 - Use case sequence diagram for SymLinks

The way that SymLink works is that it would implement a top level file in HMaster called 'SymLink Directory' that would manage all active Symbolic Links between tables and columns in HBase. In this sense, when two tables of data are Joined, they are effectively unaltered, and instead only shown as linked in HMaster. The idea can easily be seen from Figure 2, where two tables of data are Joined, but even when they're retrieved, HBase

needs to access both tables of data. This is because there is no Join ever happening in the database. Instead, when data is retrieved, the Join happens in real time before data is handed off to the user.

Advantages and Disadvantages

Immediately there are obvious advantages and disadvantages that make this idea polarizing. When presenting the concept of SymLink, it is important to remember that it is not for every database. However, because of the use case that is HBase, we will quickly see why SymLinks offers compelling advantages.

The first big advantage of SymLink is that it allows for extremely fast Join operations on datasets of any size. This is because all it takes for a successful Join operation is to add a line in the SymLink Directory stating that the two tables have been Joined. This is deceiving however as it comes with a huge disadvantage, being that retrieving Joined data is very slow as data is Joined to the user as it is retrieved in real time. This payoff makes sense when you think about HBase and its design. Being a big-data platform, a lot of tables will end up being Joined, but relatively few might be called upon, and only the popular ones will be called upon consistently. Taking this into account, moving the performance hit from the Join operation to the get operation makes sense.

The second advantage is that it is extremely easy to implement this solution in terms of a developers point of view. The fundamental parts of this Join operation are to add a new file to HMaster called SymLink Directory, and to constantly be adding and removing lines from it depending on what is considered linked. The disadvantage here comes from how Symbolic Links work. If a table is modified without any of its linked tables knowing of this, then the SymLink is broken, as the table is now a totally different table. This isn't a big deal however as, re-linking the two tables is extremely fast and easy as we mentioned earlier. Another disadvantage is that all calls to linked data will now have to be replicated among all the data locations, which can cause performance hits. We discussed this same idea somewhere with the first disadvantage, and it's a necessary evil to have.

The final advantage, and possibly one of the more important is that it will have very little impact on the architecture of HBase. The only changes that will have to be made are the inclusion of a newly added SymLink Directory folder to HMaster, and additional functionality in the methods that deal with retrieving and modifying data. The additional functionality is that the methods now have to replicate over all the linked data in different locations, which again does not impact the architecture very much. This

advantage alone makes SymLink extremely attractive, as it means easy and seamless integration with a large and imposing codebase.

Approach 2 - JoinServer

The second approach to allowing join operations on HBase databases is achieved by adding a seperate server to hold joined tables of data after they are created. This new server, referred to as a “JoinServer”, exists on the same conceptual level as the existing RegionServer module of HBase. Figure 3 shows the modified conceptual architecture.

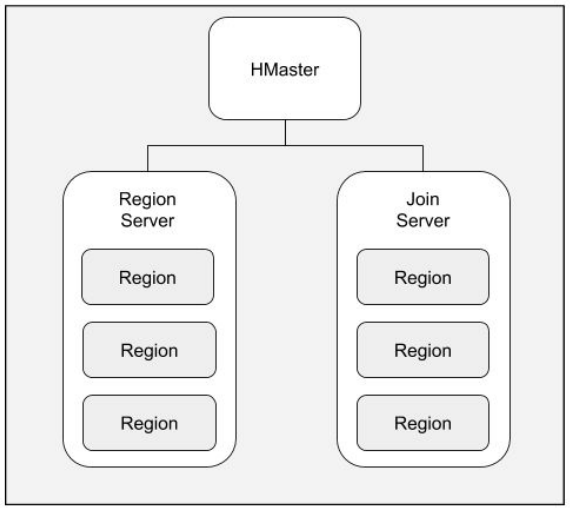

Figure 3 - JoinServer relationship to HMaster and RegionServer

When a join operation is requested the relevant tables of data will be checked against each other. Rows of data that fit the join requested will be copied and appended to form a new table stored on the JoinServer. HMaster will be responsible for tracking existing join tables and directing reads from them to the correct physical server.

Advantages and Disadvantages

The key advantage of the JoinServer approach is the improved lookup speed when compared to approach one. With a single static table stored on a seperate machine the time to look up values on the joined table is no slower than a normal read operation would be. This advantage is most apparent when the table is frequently read but does not have to be updated with new data often.

There are several disadvantages to the JoinServer approach. First, the introduction of the JoinServer module is a major change to the architecture of HBase. It would require changes to HMaster and introduce a dependency between RegionServer and JoinServer. Second, with this implementation of the Join operation there will be a significant amount of processing time required to construct the table on a separate server. Lastly, the join table would not see the information inside of it updated when the constituent tables are written to. To have the joined table reflect changed information in the original tables HMaster would have to be given the responsibility of watching for data writes and applying them twice.

Chosen Approach and Reasoning

Our chosen approach (winner) is Approach 1. We came to this conclusion based on a reasoning criteria as well as keeping the concerns of all stakeholder who would be affected by this enhancement implementation as our priority. Our reasoning criteria was based on the feature implementation itself. It includes 3 factors we compared both approaches on which are: architectural change (if the approach implies we need to change it), implementation cost and performance (for the usage operation in Joins). We have discussed each approach above and talked about the pros and cons - here is our summarized chart as shown in the table below.

Reasoning	Approach 1 (symbolic links)	Approach 2 (join server)
Architectural Change	No major architectural design change	Major change in architectural design/pattern
Implementation cost	Low implementation cost	High cost for designated server/cluster
Performance (depends on size)	Faster join operation but longer read/write	Fast for many reads and not many writes

We can see approach 1 requires no architectural change because all we need to do is add the new SymLink Directory folder to HMaster. It also cost less to implement new file to HMaster called SymLink Directory. And in terms of performance approach 1 would be faster for the join operation itself but might take longer in read/write process compared to approach 2 which can do many read and writes but slow on the actual join operation. Approach 1 wins in the first two factors and is on par with approach 2 on the third factor and hence is the clear winner. Now looking at from the views of the stakeholder. The user wants good performance which symbolic links offers. Developers would love to add a feature without the change of architecture as it would save a lot of time, effort and won't break current working system. Implementation cost is lower for approach 1 so that benefits clients (of all sizes) and also does not affect current HBase system. These are our reasonings for choosing approach 1 (symbolic links).

Effects of Enhancement with Chosen Approach

On System

The beauty of SymLinks is that it will have very few effects on most of the architecture in HBase.

Maintainability

SymLinks will not affect the maintainability of HBase at all, and will only add one more thing to be maintained, being the SymLink Directory code. Otherwise, nothing else is really changed, and as such, maintainability remains the same.

Evolvability

HBase maintains its current sense of evolvability without anything changing drastically. The only impact to evolvability would be that any changes to the methods regarding retrieving and altering data will have to be replicated to work with SymLink Directory.

Testability

Testability will not be affected in the slightest outside of adding further tests for SymLink Directory. All other existing and future testing will be unaffected

Performance

Performance will not be an issue if the Join operation is never used, however, when retrieving Joined data, the user can expect some performance hits. Outside of Joined data however, there won't be any impact to performance.

On Conceptual Architecture

In terms of conceptual architecture, again SymLink will not be changing much, and the architecture will mainly remain the same.

High level

On a high level, the architecture displayed in Figure 4 will remain the same as nothing will be impacted outside of

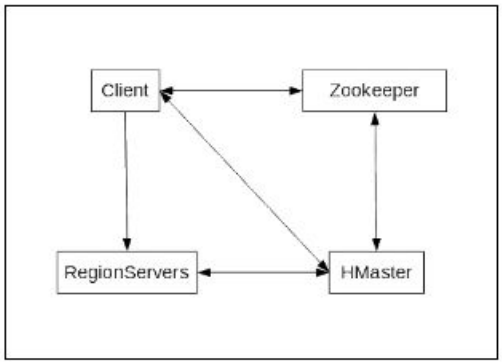

Figure 4 - Current HBase Conceptual Architecture

some files added to HMaster. Therefore at a high level, it can be expected that architecture is unimpacted.

Low level

On a low level, the changes to architecture are that HMaster will now include a SymLink Directory file, as shown in Figure 5 and will have added dependencies on RegionServers as it begins to need to retrieve information about data stores there. Outside of this small change however, low level architecture will remain the same.

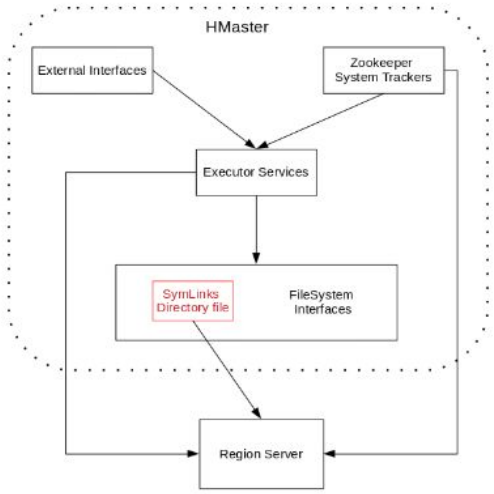

Figure 5 - New low-level architecture of HMaster with newly added SymLinks file

On Directories and Files

There will be no changes to directories and files outside of the addition of the SymLink Directory file.

Design Patterns and Architecture Styles Used

SymLink Directory will use a Singleton design pattern as it is only expected to have one instance of the directory running at any one time on HMaster. An architectural style that will be used with SymLink will be the Facade design pattern, as the relatively simple Join interface will be hiding more complex SymLink logic behind it that is very unconventional.

Interactions with other HBase Features

Insert

Implementing symbolic links will require the insert command to be changed. The key update required is to add symbolic links when a new table entry that it part of an existing join operation is added. This will ensure the joined table holds up to date table information. To achieve this HMaster will need to keep a list of existing join parameters and use them to check if incoming information should be given a symbolic link to another section of another table.

Update

The update operation will require similar changes to the insert operation. Updated table rows will need to be checked to ensure they still belong to the joined table. If not they will need to have their symbolic link to another table removed.

Delete

The delete operation can be left almost completely unchanged. The only change will be that the symbolic link for that row of data that associated it with a join operation must now also be removed.

Select

The select operation will have to be updated to follow symbolic links and construct the expected join table for the user. When the selected data does not make use of the symbolic link join the select operation will not be modified.

Compatibility Testing

Metrics

Compatibility testing of the system will focus on regression testing. There are three key metrics that will be used to check the functionality of the system: speed of read operations, speed of write operations, and maintaining scalability of the system. All of these metrics will be checked only with respect to the previously existing functionality of HBase. The performance of the symbolic links and the Join operation itself is unimportant, as any performance is better than an impossible operation.

Consistency

The read and write speed metrics measure the average time it takes to perform a large number of read or writes, respectively. The goal of these tests is to ensure the read and write speeds of HBase when not using symbolic links to perform a join operation are not impacted. The scalability test checks the speed of reads and writes as the size of the database grows. This test ensures the introduction of the symbolic links does not make HBase less scalable.

Risks and Limitations

The chosen approach, implementation of SymLinks, does not come without hindrances. Have in mind Figure 2 when trying to understand why.

As we know when creating a SymLink we only link table locations, this makes it necessary to duplicate our number of read and write operations since when we want to access Table C. Because of this it is possible to create some bottleneck situations. Meaning that when reading and writing on Table C we can considerably slow down our throughput and data flow. In addition, we will have to duplicate our number of locks when writing to Table C further explanation on the Concurrency section.

A SymLink would be a very fragile connection since it would suffice to update Table A or Table B to break the connection. We would then need to create it again. Saying this solution is best for many reads and few writes would be very accurate, since writes break the links, doing many writes counters the purpose of this solution. It that would still be viable but not as powerful.

Another concern that occurred to us was the runtime performance of MapReduce operations. Since there were changes to the logical placement of the data there is most likely an impact on this type of operations. Although without concretely implementing and testing our solutions we have no further information about this issue.

Concurrency

Keeping up with the reasoning on the last section (still using the table names from Figure 2) about having to introduce more locks when doing writes we have to imagine that, because we are technically using Table A and B at the same time, when accessing Table C instead of one lock we would now need two, some minor changes are bound to happen. Picture a situation where we are writing to Table C but the part we are modifying is only part of Table A, we would be locking Table B for no reason. This is of course a limitation of our solution but it's one with no way around.

Lessons Learned and Conclusion

HBase is definitely not ‘perfect’ in the sense that, there can still be features added on to it to make it better. There are many different approaches to enhancing on what HBase currently is, and the best one would be one that is the least intrusive on the product as it is; maintain its original functionality while improving or adding others.

3rd party applications, like Apache Hive and many others, have their own way of ‘joining’ a distributed database’s tables. Since this is built into the 3rd party software, it is not optimized for HBase itself. HBase *can* join by incorporating MapReduce but it is not that much more efficient than a 3rd party application running its own join function. Having said this, HBase having its own ‘join’ implementation that is optimized and made suited for HBase, would benefit the user, as their ‘join’ commands will no longer take a long time, and HBase, as it will already have measures taken to accommodate a ‘join’ functionality.

In conclusion, a 3rd party application *will* join correctly, but it *can* be improved on and this was the focus of the report. This improvement introduces a built-in join functionality in HBase using SymLinks. This was concluded as a couple of approaches surfaced in order to implement this feature, and both of them were analyzed based on how much they affected the current version of HBase (performance as well as interactions with other features), how beneficial were they compared to the current way of joining, and how they each affect the stakeholders of HBase. SymLinks was decided on as it showed the least intrusion on HBase currently, as well as fulfilling the ‘join’ functionality, with its own fair of drawbacks such as increased lookup time. Having decided on this, this enhancement meets all of its requirements as mentioned in the abstract.

Appendix

A: Naming Conventions

B: Bibliography

[1] https://www.google.ca/search?q=Dictionary#dobs=enhancement

[2] https://kb.iu.edu/d/abbe