BROADPEAK PARTNERS, INC.

Big Table in Plain Language

Some people remember exactly where they were when JFK was shot. Other people remember exactly where they were when Neil Armstrong stepped on the moon. I remember exactly where I was when Google began to populate real time search results as you typed. I was with a team of developers at a large utility in the Midwest working on a large project. One of the developers started screaming, "Oh my God! Look at this …Google has real-time results." A small crowd gathered around his desk while he typed in letters and started to get real-time results. That was 2005.

We take it for granted now. It is so…been there done that. But the technology behind what makes this type of functionality possible is the result of a landscape change in how data is managed. Enter the No SQL movement…aka "Big Table."

Big Table is a data storage concept that Google hatched onto the world stage back in 2004. To a certain extent it has been supplanted with its newer cousins such as Hadoop, Cassandra, MongoDB and others. If you read the recent press there is a lot of hubbub about the NOSQL movement. NOSQL is just another name for the original Google Big Table concept. So what is it about Google, Facebook, Amazon, Yahoo and many, many others that have literally bet the house on Big Table functionality?

To understand this landscape shift in data use, we have to go back to where we started: the Relational Database. Even those hiding under a technology rock know a bit about these databases including Oracle, MS SQL Server, Sybase and others. The impact Relational Databases had when they hit the ground running in the late 80s was massive. Oracle built this one simple technological concept into a $145 billion dollar company.

So let's begin where we started: Pre-relational database. Old school databases looked like this:

Employee Name	E-Number	Department	Department Number
Vivek Pathak	22	Marketing	10
Manish Handa	10	Development	2
Gordon Allott	11	Sales	1
Tom Cheng	33	Development	2

Here is a list of employees, their employee number, department, and department number. If we decided to delete Tom Cheng and Manish Handa from the database we would also, inadvertently, delete the Development department in its entirety. If after deleting these two we were to query from our application of who is in Development, the application would error out because *there would no longer be a development department*. Enter endless work-arounds to keep this from happening.

Then along came the Relational Database. This structure split key information into separate tables. For example in a relational database we have:

Employee Name	E-Number	Department
Vivek Pathak	22	10
Manish Handa	10	2
Gordon Allott	11	1
Tom Cheng	33	2

Department	Department Number
Marketing	10
Development	2
Sales	1

This structure allows key information to "live" in separate tables. Key tables are then "related" to each other through a function called a "Foreign Key." With the two tables related, it is a very simple matter to create "joins," where data form one table is joined with another. This is the magic of a Relational Database. To a human, this seems less efficient, but to an application this ensures that key data is maintained consistently. We often work with trading applications that use Relational Databases like Oracle. It is not uncommon at all for a trading database to have well over 1000 tables, even for something simple like a trade. There might be 20 or 30 separate tables used to represent the static reference and trade term data required to book a trade.

So what's the problem? Using data in calculations, that's the problem. Let's say that instead of having 4 employees, I have 400,000. I need my application to execute some process across each of those employees like for example, assigning a year-end bonus. So I create some logic that pulls each employee, its group and perhaps other data like years of service with the firm and execute a calculation routine that allocates a bonus to each employee. This requires my calculation logic to do a database query pulling data from each of its respective tables. Is this a problem? No, it's not a problem at all. But with larger data sets and a large amount of tables this gets very costly in terms of time. Perhaps as much as 80% of the time needed to perform the overall calculation is spent on database calls. These get slowed down by hard drive performance (read /write rates) and having to travel over the network.

To give you a practical scenario, we were working with a trading house that was unhappy with the end of day "mark to market" calculation time. This is a process that calculates the final value of each trade according to closing prices. Over 50,000 live trades it took the calculation 6 hours. They tried adding more hardware, but it had little effect. When we investigated the trading application we found that for each of the 50,000 trades, the application was making a total of 16 database calls to Oracle. So, that equals 800,000 database calls each taking about 2 milliseconds per call. Forget the mark to market calculation; this client is spending over 266 minutes, or 4.4 hours just getting data *to the calculation*.

In short, large datasets get difficult with Relational Databases. For a company like Google, this would have been a company killer. If users are anything, it is impatient. Google searches the web and "indexes" all of its web-pages. Each has unique data that if held in relational tables would take an enormous amount of time to search if they sat on a relational database. Sure, you could throw hardware at the problem, but would it be

worth it? In the absence of something better consumers would really be looking at "pay for search" which is really just paying for all the hardware to cull the web and return results in a timely manner.

So Google needed a total paradigm shift and something entirely new to create efficient database use. What they came up with is Big Table.

Big Table data structures work a bit differently. The first thing is how the tables look and act. Remember the tables above? We have rows and columns. Each column represents a data parameter; each row represents the population of data fitting that parameter. What big table essentially does is condense all of that columns and row data and packs it into rows. We end up with something that looks like this. Each row has all of the (formerly relational data packed into it). What used to be separate tables are now grouped into "families." This allows us to get all formerly relational data into a single row of a single table. We can now throw heaps and heaps of data into what is essentially one gigantic table or naturally: BIG TABLE.

Ref Key	Employee Family	Department Family
Ref Key 1	Vivek Pathak|22 | 10 |	Marketing | 10 |
Ref Key 2	Manish Handa | 10| 2 |	Development | 2 |
Ref Key 3	Gordon Allott | 11 | 1 |	Sales | 1 |
Ref Key 4	Tom Cheng | 33 | 2 |	Development | 2 |

So when a user wants to find out every employee in the Development department it is a simple call to the database using the term "Dev." What is returned is all the reference keys that have "Dev" within the row. It is a much more efficient way to query big data sets.

The First Big Table "Trick"-Partitioning

The first Big Table trick has to do with the enormous potential size of the table itself. It is called a "BIG" table after all. But big equals problems in terms of processing. When you have a single enormous table and need to find, say, all people in the development department it requires searching one very big and long table. Think about what Google is using this for. It is an index of every website on the Internet. Key terms, phrases and data are captured and stored to a single big table. We are not talking about a table with megabytes, gigabytes or terabytes of data, but a big table potentially in the petabytes (1 Million GB), Exabyte (1 Billion GB) and beyond. This is a huge storage problem. Think about the size of a server necessary to store an Exabyte of data. It would take up a football field and that is just to do the storage. So Google came up with something very clever to avoid the super-massive sever: Partitioning.

Partitioning is very straightforward. Basically you take let's say 5 of some big but off the shelf servers. The data in the Big Table is partitioned across each of the 5 servers. So for example if you had a 1 TB big table, each of the 5 servers would have 1/5 of the table. This makes the data storage much more manageable, because you can execute big storage across a whole number of off the shelf servers. Big Table automatically distributes the table over whatever server space is made available to it. It also flows very nicely into the second Big Table trick: Map Reduce

The Second Big Table "Trick"- Map Reduce

Remember how one of our challenges with marking 50,000 trades was that we had to pull all of this data and push it to the calculation component? Big Table allows us to do something different. Instead of *pulling* the data to a calculation component, we can actually "drop" the calculation directly into the database to calculate there. This totally eliminates the kind of database calls that had slowed us down in the first place.

Ref Key	Employee Family	Department Family	Results
Ref Key 1	Trade 1	Swap | Fixed Income	Jan 1 | 3.4 | Jan 2 | 3.3|
Ref Key 2	Trade 2	Swap | Fixed Income	Jan 1 | 3.9 | Jan 2 | 3.3|
Ref Key 3	Trade 3	Swap | Fixed Income	Jan 1 | 3.41| Jan 2 | 3.3|
Ref Key 4	Trade 4	Swap | Fixed Income	Jan 1 | 3.1 | Jan 2 | 3.3|

Here all we need to do is create our calculation logic "pill" and drop it into the database where it will calculate and save results in each row. If you think about this, it is a very efficient model for companies like Google to use. Its famous web crawler searches the web, looking for updates, key terms, links, and all other sorts of data over time. This gets efficiently stored in the Big Table. So when you start to type a word and get real time results back, it is because each time you type a letter it is dropping that search pill into the Big Table to pull results.

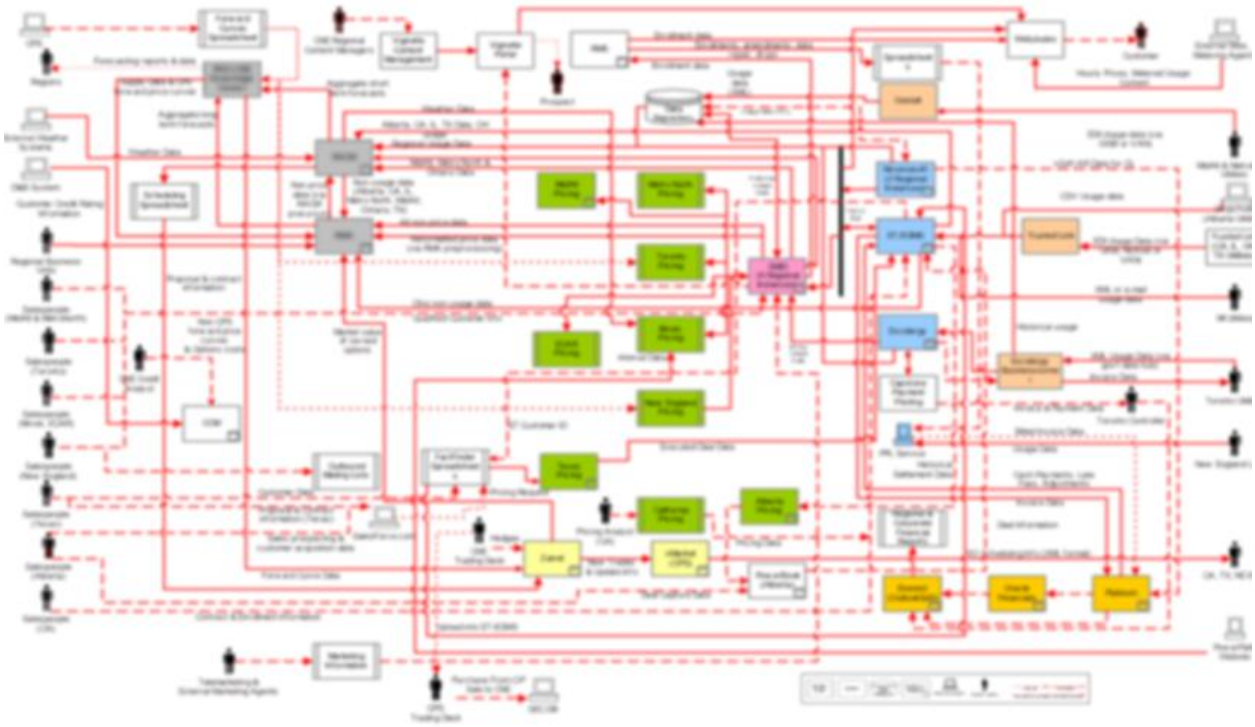
Map Reduce is really the mojo that makes Big Table work so well. With partitioning I have broken up the Big Table into multiple chunks. Each chunk runs on one of a cluster of servers. So, one might think about this gigantic table but living on 1000 servers at the same time. With Map Reduce when I need to execute a calculation, the Map Reduce actually sends the calculation pill to each chunk of the Big Table simultaneously. For example a Big Table might be 1 Terabyte consisting of row after row of data. But what Map Reduce can do is partition that data over, say 10 servers. Each server has a partition of $1/10^{th}$ of the overall table. Through Map Reduce when I drop my calculation pill, it actually distributes that calculation to each chunk of the Big Table on each server so that it is being done simultaneously Results? Nearly instantaneous. To a certain extent this is what is driving a lot of hubbub about the "Cloud." You can put these large data sets into absolutely gigantic server farms where you can Partition and Map Reduce to your heart's content.

So does this mean that Relational Databases will go away? Hardly. Relational Databases have two realistic advantages. First and foremost is that relational databases are very well understood and have a massive pool of knowledgeable users. Most applications are built on relational databases and there will always be a very big need for relational data structures. Second, there are always going to be times where a relational database is a better option than using some type of Big Table technology. Without getting into too much detail, the key asset that relational databases provide is consistency. When exact data is mission critical developers will opt for a relational database. For example, bank accounts. But where Big Table really gets its

legs is when we need to have flexibility searching, data mining, and executing functions across very large datasets. What do we think this means?

The Truth About Enterprise Data

The truth about the average business enterprise is that data is pigeon-holed across a cast of good (read SOA) and badly designed applications. And each of these applications is tucked behind what are usually custom interfaces. This is the Jersey Barricade sitting between "the Cloud" and enterprise data. Take for instance this interface diagram:



I wish I could tell you that this was not typical. But in fact this is your typical interface diagram for a midsize division in a midsize enterprise. Each red line represents an interface point. If you wanted to get this enterprise data into a Big Table you've got to run the integration gauntlet.

This is one aspect of enterprise integration we are keenly focused on with K3. We believe that K3 is a key element to extracting data out of older architectures and replicated in a Big Table format. If it is on the fly calculations, searches, and other data mining activities, Big Table will take the enterprise a long long way. But we have to focus on getting the data there in the first place. So that is what we are doing. We replicate data flowed through our Interface Harness into a No SQL database. What will we do with it from there? Well we are only limited by our imagination.