# a database pardigm;TRILOGY

Walter B. Gress V

Drexel University

Friday, May 27, 2016

Databases are notorious for being slow, clunky and difficult to integrate into the rest of

the applications.This report introduces a better, faster database and included with that

replace a better replacement SQL.

# introduction

This is a primer on an database accelerator I’ve cleverly named Trilogy. Given a series of SQL-like queries (TQL) resulting in a success, with clustered together. I propose a significant improvement over SQL.As I said in the abstract, SQL and the whole database construct could use revision. Implementing this SQL-like queries is client side.

# lxcache

Fragmented DB memory. I am assuming the project has some form of defragmenting. First principle is a form of automatic defragmentation. There are a combinations of architectures.

This architecture is designed specifically for a database that is heavy hitting, with a large volume of queries rapid enough that in a window of time that the client side slows the transaction time.

Resources are distributed according to cache hits and misses in a manner similar to a core of a CPU;In this case it is the Trilogy Central Core (TCC) which processes the query and caches.

Buffer In

TCC

Buffer Out

One way to accelerate a database would be to use a Database Processing Unit, that is, an on board database accelerator. We presently have huge server farms, especially google, and I would assume Netflix and many others. The DPU would be a client side chip that would do various things which I will outline in the rest of this paper.

The Trilogy query is passed to the main buffer (more is better). The caches are memory model of the way a CPU and L1, L2, etc. cache work.

Trilogy uses a multi-level cache. If the computer misses the outermost cache then it will move to the next level, and so on until it reaches the BUFFER IN. This buffer is implemented with a Hashtable and is endless.

When data is required, the L1 cache is hit. If similar data is required the L2 cache is hit. If similar data is required the L3 cache is hit. Finally the caches are exhausted so TCC fetches from Buffer In.

Generally, the trend is to keep the cache closest to the procesor the fastest and thus the smallest, being the most expensive.

The Trilogy model segments memory resources and CPU resources. A query is passed to the buffer. The key to Trilogy is buffering and reuse of common data. This all happens client-side.

Trilogy is queried and return the data from the query as well as leave markers. The markers store the tracking history.

# FETCH/return blocks

Fetch command sent from CPU to DPU.

DPU fetches the query data in the form of a block of data and the TQL and stores it until the operating system is ready for it.

The return block being returned consistent of the original query, and the payload.

# hash function

Hash Functions are used to cut back the resources used for disk access. Given such as:

1. TQL Query A
2. TQL Query B
3. TQL Query C
4. TQL Query B
5. TQL Query C
6. TQL Query C

Given those TQL queries, and asked where to logically start, the best bet to examine the clusters and it is obvious the choices the C. Again, the processing sits on the client chip.

# Trilogy rle

Trilogy uses an altered form of the classic Run-Length-Encoding algorithm. The database is compressed (Server Side). The database is monitored and dynamically checks for slack in the database. Weak areas are compressed to make space . When the client requests a query, the returning block is going back compressed. This is another benefit to compressing the data. There is less data to process. When the query results reach the client TCC, the TCC decompresses the results and the data is routed via the TCC to the Control Bus.

# CODE

Ultimately we are dealing with a client-server architecture. The cache cascade could be either or both. The Hash Function is a client or server – toss-up. The compression algorithm is obviously a server algorithm. All of this comes together to form a single architecture. The database operates on a light-weight assembled code base.

The DPU is programmable. Similar to FPUs that brought new functionality back in the 1990s and the GPU that not that long ago brought amazing graphics. I propose we follow suit. Imagine Big Data. Imagine the speed we would get streaming. Imagine even Google could make out as if they didn’t already have a server farm the size of a small island.

This is excited it. You may be a naysayer about the algorithms, the DPU is a solid idea.

# ASSEMBLY

DPU is a solid idea. We agree it is. It’s not sexy GPUs or necessary like a FPU. But let talk about a delivery. First location. There is no point in putting it in the server, we already are stuffing those things chips like this already. Second, the code. The TQL code is compiled code byte-code See Table 1.1 for the instruction table.

Third See Table 1.2 for the pipeline architecture.

TCC (4 ADC multicore)

ADC -> Analytical Database Core

(actualf uncitonal core is TDU)

TDU (Trilogy Database Unit)

**Acts as a buffer until it can be processed**

# lexicon

We have a programmable device, so we need language to program it. Again, the DPU takes the pogram and assembles it and the compressed, assembled query is sent over to the server. There is a speed gain where the server reads the assembled code natively. This may not seem to be worthwhile but even a thousand or tens of thousands queries it adds up. Now I’m going to demonstrate the DPU instruction set,

**main entry point to a query**

**array a temporary array to store results from query shelf-life is scope**

**iterator for iterator=0;iterator<10;i++**

**{ array[iterator] = tablename[iterator][\*]**

**tablename the name of a table**

**table a temporary table (collection of temp arrays) shelf-life is scope**

**for see above under iterator. Executes a for loop**

**while**

**Int x //temporary variable**

**Targetnumber //sentinel variable**

**while(itereator < targetnumber)**

**{ x++ }**

**return x**

**if if (table\_name[5][5] == 10)**

**{ double x = Col(first=0, last 10) }**

**else { double\_x = Row(first=10, 20) }**

**{ return double\_x}**

**Else Else is the command for the opposite line of execution**

**return(x) Complete the query and store the results in x**

**with Jump to a different scope temporarily**

**print “Hello World” (a is not in scope)**

**with { other\_object }**

**{ print(“Hello a”) (a is in scope) }**

**With is over, a is no longer in scope**

**var function prototype function:**

**myreturnname function(param1=value,param2=value,param3=value,param4=value)**

**function call: myFunctionName(Cigars=10, RumBottles=5, CasesOfWise=10)**

**int 4 bytes**

**double 4 bytes**

**float 8 bytes**

**big 64 bytes**

**char 1 byte**

**string 8 bytes variable**

**create database() *create\_database database\_name***

**drop database() *drop database\_name***

**create\_table *create\_table(int=”whiskey”, string=”J.R.R. Tolkien”) The parameters are the names of the column table.***

**fetch**

works like sql select where retrieves a column

**where**

works like sql where parameters are put into the SQL query

**resolve**

turns a string into a cell in the table and turns a cell into a string

**[][]**

**Table (array) operator**

Stores/Retrieves the table into a variable

array\_y[5][5] = 10 arrray\_y Row(5), Col(5) gets the value of 10

x = array\_y[5][5] Retrieves the value of table [5][5] and stores it in X

**Col(first=value, value=last)**

Searches from a column from first to last index

**Row(first=value, last=value)**

Searches from a row from first to last index

**Span(column=value, table=value)**

Searches the entire table

**Update(value=Juan, table=value)**

Updates the entire table

**InnerJoin(on=value, strip=value, strip=value, …)**

The on parameter is the match of the strips values

**\* Wildcard**

**+ One or more**

**The following is TQL and the SQL comparison.**

***select \* from table\_name > selects all of the table***

**int x[][] = fetch \* my\_table**

***select \* from table\_name where country = Mexico > selects all the Mexicans in the customers table***

**int x[] = fetch [] \* customers\_table where country = Mexico**

***update Customers set contact\_name=’Juan’ where CountryName = Mexico***

***updates are customers for all records with the country name is Mexico***

**update(Juan, Mexico)**

***insert into customers (Customer Name, City, Country) Values (Cardinal, Stavenger, Normay)***

**insert(Customer Name, City, Country) => values (Cardinal, Stavenger, Norway)**

***select column from table inner join table 2 on table 1 column = table 2 column***

fetch \* Col(column\_one) => (column\_two)

Each **s** chip is see-saw. That is, they can alternate functionality. For example if it

needs more power on the dedicated chip then the **s** bus requests and the chip changes functionally temporarily. The control bus handles routing the data.

**s-chip**

**s-chip**

**s-chip**

**s-chip**

**s-chip**

**While**

**Do while**

**If**

**Else**

**Return**

**Int**

**Float**

**Double**

**Big**

**With**

**Var function (**

**s-chip**

**Else**

**Return**

**Int**

**Float**

**Double**

**Big**

**With**

**Var function (**

Control Bus

S Bus

File the s-chips can flip-flop to are specific instructions:

**fetch, where, resolve, [] operator, Col, Row, Span, Update, InnerJoin**