EFFICENT QUERY PROCESSING METHODS

POOJITHA GORANTLA
Masters in computer science

poojithagorantla@my.unt.edu

Abstract

The Data server has a subcomponent called the query processor which processes SQL requests. Using SQL requests, we can access a single database or multiple databases from vast volumes of data. In this paper, we have mentioned many methods to increase the speed of accessing the data. The aim is to optimize the effectiveness and efficiency of the process where users communicate to the system regarding data they intend to extract from the database. Moreover, the readability and comprehension improvement of the query visual representation is intended, reducing the time and the effort required to understand what data will be gathered from the database.

Keywords

SQL, arrays, Spark, NoSQL databases, DOE, CUDA, GPU, Visual Query Interfaces, Big Data

1. Introduction

Nowadays, database queries are required not only in computer systems areas but also in most sectors of professional or personal environments. The increase of the database information gathering needs, increased the searching for advanced technologies to optimize the time spent and reduce the errors of this data querying process since obtaining the intended information is most important.

We have seven methods in total that we have mentioned in the paper that have used for efficient query processing. In paper 1 a novel framework, called SQLgen, that automatically translates imperative programs with loops and array operations to distributed data-parallel programs. Unlike related work, SQL gen translates these programs to SQL, which can be translated to more efficient code since it can be optimized using a relational database optimizer. SQLgen has been implemented on Spark SQL. We compare the performance of SQLgen with DIABLO, hand-written RDD-based, and Spark SQL programs on real world problems.

In the second paper they proposed a hardware-software co-designed system, DOE, which contains hardware accelerator architecture - Conflux for effective SQL operation offloading, and a software programming platform - DP2 for application integration non-intrusively.

In the third paper they utilize the Oracle partitioning technique to enhance specific query performance. Physical database tuning approaches of SQL databases are being used to speed up each SQL statement such as partitioning and indexing.

In the fourth paper A new method for SQL query implementation in a parallel execution environment Apache Spark in a package mode was developed based on Bloom Filter Cascade Application (BFCA). It includes representation of the original query in a form of a few subqueries and intermediate tables, where the Bloom filters are created and applied.

In the fifth paper computational intense fields are increasingly using CUDA to deliver the benefits of GPU acceleration. Also, CUDA is used in boosting the performance of database-oriented problems, speeding up the SQL performance on a huge dataset resulting in acquiring the data in rapid speed. This paper focuses on reading data from a .csv file, implementing kernels and performing the basic SQL aggregation functions on the GPU, and comparing their performance with the MySQL counterpart, showing the worthiness and importance of the concept.

In the sixth paper they implemented interface is currently incorporated on the OutSystems Platform to accelerate the query formulation process without harming the learnability of the system. The users' interviews and the results of the SQL queries evaluation have foreseen that the cause of the reduced acceptance of the visual approach could be the existing usability problems on the interface. Furthermore, the interface is inadequate to build more complex queries, which involve more entities and conditions.

In the last paper an indexing structure which is a pluggable component of Spark SQL based on Apache Spark. Compared with Spark SQL, it has some additional advantages. Firstly, it allows users to create index of structured data to be processed, which speeds up the query performance greatly. Secondly, it enables programmers to load fine-grained data file of structured data into memory, which is flexible to load "hot data" into memory and to evict "cold data" out of memory

2. Description and Technique

2.1 Translation of Array-Based Loops to Spark SQL

We have designed a framework that will translate array based loops to a declarative domain-specific language (DSL), more specifically, Spark SQL . At first, loops are translated to equivalent monoid comprehensions and then to Spark SQL. Not all loops can be translated to SQL. We provide simple rules for dependence analysis that detect loops that cannot be translated to SQL. One such case is when an array is read and updated in the same loop. For example, we reject the update V [i] := V [i − 1] + V [i + 1] inside a loop over i because V is read and updated in the same loop. But, unlike most related work, we can translate incremental updates of the form V [e1]+= e2, for some commutative operation + and some terms e1 and e2. We chose Spark SQL as our target language since it is in general more efficient than the Spark core API because it takes advantage of existing extensive work on SQL optimization for relational database systems. In Spark SQL, datasets are expressed as Data Frames, which are distributed collection of data, organized into named columns. The schema of a Data Frame must be known, while Data Frame computations are done on columns of named and typed values. Operations from the Spark Core API, on the other hand, are higher-order with arguments that are functions coded in the host language and compiled to bytecode, which cannot be analyzed during program optimization. Hence, Spark SQL can find and apply optimizations that are very hard to detect automatically when the same program is written in the Spark Core API. Spark DataFrames have two specialized back-end components, Catalyst (the query optimizer) and Tungsten (the off-heap serializer), which facilitate optimized performance on other Spark components, such as MLlib, that are primarily based on DataFrame API. Catalyst supports both rule-based and cost-based optimization. For example, it can optimize a query by reordering the operations, such as pushing a filter operation before a join operation. The operations in Spark SQL reduce the amount of data sent over the network by selecting only the relevant columns and partitions from the dataset necessary for the computation. Consequently, we expect that loops translated to Spark SQL, as in our framework, would perform better than loops translated to Spark RDD operations, as it was done by

earlier frameworks.

2.1.1 Program Translation

We translate the comprehension in two steps: pattern compilation and comprehension translation. Pattern variables in a comprehension are defined in the generators and used in the rest of the comprehension. However, SQL does not support patterns. To address this problem, we eliminate patterns by substituting each pattern with a fresh variable and by creating an environment ρ that binds the variables in the pattern to terms that depend on the fresh variable. The fresh variable is also used as the alias for the SQL table. For example, if there is a generator ((i, j), v) ← e in a comprehension, we replace it with x ← e, where x is a fresh variable, and we create an environment ρ = [i → x. 1. 1, j → x. 1. 2, v → x. 2], which expresses i, j, and v in terms of x. (The term x. n returns the nth element of the tuple x.) Given a term x and a pattern p, the semantic function CJpKx returns a binding list that binds the pattern variables in p in terms of x such that p = x:

C[[(p1, . . . , pn)]]x = C[[p1]]x. 1 ++ · · · ++ C[[p]]x._n (1)

C[[v]]x = [v → x] (2)

where ++ merges bindings. For our example, after applying (1) on ((i, j), v) we get the bindings:

C[[((i, j), v)]]x = C[[(i, j)]]x._1 ++ C[[v]]x._2

= C[[i]]x._1._1 ++ C[[j]]x._1._2 ++ C[[v]]x._2

Then, applying (2) we get, CJ((i, j), v)Kx = [i → x._1._1, j → x._1. 2, v → x._2]. Before the translation to SQL, we eliminate the patterns from a comprehension as follows. For each generator p ← e 0 , and any sequences of qualifiers q1 and q2 in a comprehension, we do:

{ e | q1, p ← e' , q2 } = { ρ(e) | q1, x ← e' , ρ(q2)} (3)

where x is a fresh variable and ρ = C[[p]]x, which expresses the variables in p in terms of the fresh variable x. The ρ(e) and ρ(q2) replace all occurrences of the variables in e and q2 using the binding ρ. For example, the comprehension

{ (i, a + b) | (i, a) ← A,(j, b) ← B, i = j }

is translated to:

{ (x._1, x._2 + y._2) | x ← A, y ← B, x._1 = y._1}

The next step is the translation of a comprehension to SQL using the semantic function SQL, which takes the comprehension as input and translates it to a Spark SQL query:

SQL[[{ h | q }]] = select h from Q[[q]] whereP[[q]] group by G[[q]] (4)

where h refers to comprehension head and the semantic functions Q, P, and G translate a list of qualifiers to SQL tables and joins, predicates, and group-by expression respectively. They are described next in this section. The comprehension head h is translated to a select clause. For a total aggregation over a comprehension, such as ⊕/{ h | . . . }, the monoid ⊕ is applied to the translation of the header h in the select clause. For example, the header of +/({ v | (i, v) ← V }) is translated to select sum(v). We use the semantic function Q to translate the generators in a comprehension to SQL from clauses. If there are pairs of generators in the qualifiers correlated with a join condition, we translate each such pair to a join clause along with a join condition. In the following rules, the semantic function Q takes a list of qualifiers as input, identifies joins, and creates a SQL join clause with a join condition:

Q[[q1, v1 ← e1, q2, v2 ← e2, q3, e3 = e4, q4]] =

Q[[q1, (v1, v2) ← (e1 join e2 on e3 = e4), q2, q3, q4]] (5)

where e3 = e4 must correlate the variables v1 and v2, that is, e3 must depend on v1 only and e4 must depend on v2 only, or vice versa. The remaining generators are translated to table traversals:

Q[[v ← e, q]] = e v, Q[[q]] (6)

Q[[e, q]] = Q[[q]] (7)

Q[[ ]] = ∅ (8)

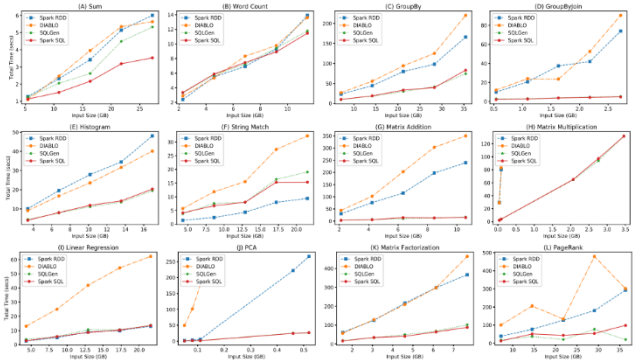
where (6) collects the generators that are not joined with any other table as simple table traversals. The semantic function P is used to collect condition qualifiers. It takes a list of qualifiers and translates them to a list of SQL conditions:

P[[e, q]] = e and P[[q]] (9)

P[[p ← e, q]] = P[[q]] (10)

P[[ ]] = ∅ (11)

The semantic function G collects the group-by keys. Currently, our



translation algorithm accepts at most one group-by qualifier. G takes a list of qualifiers as input and returns an optional group-by key:

G[[group by p, q]] = p (12)

G[[p ← e, q]] = G[[q]] (13)

G[[e, q]] = G[[q]] (14)

G[[ ]] = ∅ (15)

Examples of Program Translation

Consider a loop-based program that sums up the values of an array, written as follows:

sum := 0

for i = 1, 10 do sum += V [i]

Here, the values of an array V are summed and assigned to a variable sum. The comprehension of this program is:

sum := +/({ v | (i, v) ← V, inRange(i, 1, 10)})

where the predicate inRange(x._1, 1, 10) returns true if 1 ≤ x. 1 ≤ 10. Before the translation to SQL, we eliminate the patterns from the comprehension. The only pattern in the comprehension is (i, v), which is replaced with a fresh variable x. Then, using (1) and (2) we get C[[(i, v)]]x = [i → x._1, v → x._2]. Therefore, using (3), the comprehension is transformed to:

sum := +/({ x. 2 | x ← V, inRange(x._1, 1, 10)})

To generate the equivalent SQL query, we use (4) to translate the transformed comprehension to SQL where the semantic functions take the qualifiers of the transformed comprehension as their input. select sum(x. 2)

from Q[[x ← V, inRange(i, 1, 10)]]

where P[[x ← V, inRange(i, 1, 10)]]

group by G[[x ← V, inRange(i, 1, 10)]]

= select sum(x. 2)

```
from    V x
where   1 <= x._1 and x._1 <= 10
```

As another example, consider the matrix multiplication between the matrices M and N, which is stored in the matrix R:

```
for i=0, 10 do
    for j=0, 10 do {
    R[i , j ] := 0.0;
    for k=0, 10 do
        R[i , j ] += M[i,k] *N[k, j ];
```

The comprehension of matrix multiplication is as follows:

R := { ((i, j),(+/v)) | ((i, k), m) ← M,((k 0 , j), n) ← N, k = k' , let v = m ∗ n, group by (i, j)}

To keep this example simple, we omit the inRange qualifiers. In the comprehension above, the patterns ((i, k), m) and ((k 0 , j), n) are replaced with fresh variables x and y. Then, after applying (1) and (2), we get the following bindings:

C[[((i, k), m)]]x = [i → x. 1. 1, k → x. 1. 2, m → x. 2],

C[[((k 0 , j), n)]]y = [k 0 → y. 1. 1, j → y. 1. 2, n → y. 2].

Then, these patterns are eliminated using (3) and the comprehension is transformed to:

R := { ((x. 1. 1, y. 1. 2),(+/v)) | (x ← M, y ← N, x. 1. 2 = y. 1. 1, let v = x. 2 ∗ y. 2, group by (x. 1. 1, y. 1. 2)}

Then, we can get the equivalent SQL from (4). In the select clause, we get, x._1._1, y._1._2, sum(x._2 ∗ y._2) where v is substituted by the let-binding expression. Next, the semantic function Q is applied to the transformed comprehension in the from clause. Using (5-8), we get:

```
Select      x._1._1, y._1._2, sum(x. 2 ∗ y. 2)
From        Q[[x ← M, y ← N, x. 1. 2 = y. 1. 1,
            group by x._1._1, y._1. 2K
group by    G[[q]]
 = select   x._1._1, y._1._2, sum(x._2 ∗ y._2)
from        M x join N y on x._1._2 = y._1._1
group by G[[q]]
```

Next, we apply the semantic function P to the transformed comprehension in the where clause, which is not shown here. Then, we apply semantic function G to the transformed comprehension in the group by clause. Using (12), we get:

```
select      x._1._1, y._1._2, sum(x._2 ∗ y._2)
 from       M x join N y on x._1._2 = y._1._1
group by    G[[x ← M, y ← N, x. 1. 2 = y. 1. 1,
            group by x._1._1, y._1._2]]
 = select   x._1._1, y._1._2, sum(x._2 ∗ y._2)
from        M x join N y on x._1._2 = y._1._1
group by    x._1._1, y._1._2
```

The final translation is an assignment that assigns the result of the generated SQL query to table R.

2.2 DOE: Database Offloading Engine for Accelerating SQL Processing

The computing tandem of CPU-Accelerator platform, as a typical heterogeneous system, is believed to be a promising paradigm to unlock the advantage of domain-specific computing.

1) Integrating an accelerator into software application requires massive and intrusive developments.

2) Accelerator customization is highly complex.

This paper demonstrated that integrating an accelerator without intrusive modification into database is possible

A. The potential of database offloading

1) CPU-ACC Architecture:
The three layers in a typical heterogeneous computing architecture.
i) Device,
ii) Device Driver
iii) Application

2) The SQL execution flow of database system: includes two stages, Planning Stage and Execution Stage.

The query plan is a binary tree structure, where the tree nodes are the query operators. Execution stage computes the result by executing plan tree recursively.

3) The opportunity of database offloading: offloading solution is systematic engineering of database applications, drivers and device development.

B. Offloading-friendly SQL Operations

1) Selection & Projection
2) Group-by & Aggregation
3) Hash Join
4) Sort

The above operations can be classified into two categories: i) **scan-like**, continuous memory access, including selection & projection, and aggregation & group-by; ii) **joinlike**, random memory access, including hash join and sort.

C. The "Tricky" DRAM bandwidth for DOE customization
DRAM bandwidth optimization becomes a key breakthrough for performance improvement.

2.2.1 DP2: DOE PROGRAMMING PLATFORM

The platform aims to enable the programming capability and computation offloading, The boundary between DOE platform and database applications is the query plan and operators.

A. Operator Interfaces We design the operator interface based on the concept from open-source databases.

B. PlanTree Interface The query plan is a binary tree, in which the tree node is the Operator Interface.

C. Common Function Lib The common function lib provides the column-oriented operation functions based on the device capabilities, and is used for the implementations of operator interface by application developer.

D. DP2 Compiler The DP2 compiler is built for compiling the offload-able programs to device instructions based on the DOE instruction set.

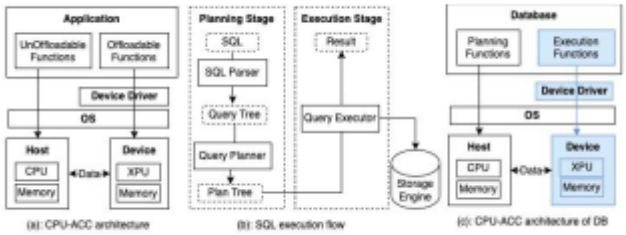
E. DP2 Runtime Lib The DP2 runtime lib provides runtime APIs for applications to execute the offloaded query plan

we evaluate the proposed DOE system in terms of DRAM bandwidth utilization, performance, and resource consumption.

1) Baselines: All evaluations were conducted on a Dell Precision 7920 tower server featuring two intel Xeon Gold 5218 CPU @2.30GHz, with 64GB of memory (four 64-bit 16GB DDR4-3200)

2) Benchmark: We used the TPC-H dataset as the test data. considering communication constraint, we evaluate each type of P-Core isolately to bound the maximum number. The DRAM bandwidth utilization is a key metric for evaluating the database system. DOE stably achieves over 60% DRAM bandwidth utilization across most of queries. DOE provides more

than 100x performance speedup over PostgreSQL, and 10x over MonetDB across most queries, except q9 and q18.



(a): CPU-ACC architecture (b): SQL execution flow (c): CPU-ACC architecture of DB

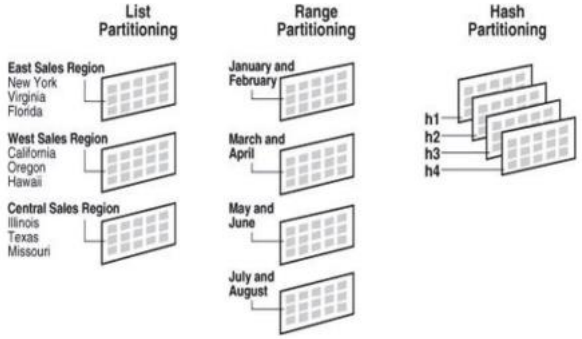
2.3 Robust Partitioning Scheme for Accelerating SQL Database

The database physical model is called a physical database design. The process of transforming the logical design and data into the optimized physical structure is known as physical database tuning or schema tuning.

Oracle Ilg Tablespaces: all the users' table data is stored in a single tablespace in Oracle 11g. The Oracle 11g database performance can be improved by creating user defme tablespaces for large and transactional tables of a particular schema. The data files are distributed into small chunks. These small chunks are known as extents and the default size of a single extent is 64k. Further, then extents are divided into small blocks and are called data blocks or oracle blocks. The block size can be 2k, 4k, 8k, 16k, and 32k. Default block size is 8k.

Oracle 11g Partitioning: Many applications performance, availability, and manageability can be enhanced through partitioning. Via partitioning 2 [15] the following database objects can be subdivided into a number of small chunks and pieces. Each chunk is known as a partition of a particular database object. The database objects are tables, index-organized tables, and index. There are many partitioning techniques provided by the Oracle 11g enterprise edition. Range Partitioning: is the most common and is used on a range of values.

Hash Partitioning: is based on the hashing algo- rithm and distributes all the records of a particular table among all the hash partitions. Over the range portioning it is easy to use. Particularly, it is more suitable where there is no historical data or not has the appropriate partition key. List Partitioning: is partitioning the rows explicitly based on the discrete list of values. Through list partitioning, we can organize and group the unordered and irrelevant data in a natural way.



2.4 Bloom Filter Cascade Application to SQL Query Implementation on Spark

NoSQL databases have high horizontal scalability (thousands of nodes) and overcome some RDB limitations. They have limited applicability boundaries and also depend on the node RAM volume. Bloom filter [10] is an array of bits built on the table key hash values. This filter misfires with 1/2H probability, where H is a Bloom filter parameter. With a significantly high H value, very low false-positive value probability is achieved. This reduces the records number in a join, occasionally fully eliminating the join. This significantly improves query processing time.

This analysis demonstrates that SBJ and SBFCJ data warehouse access methods reviewed in [8] have advantages. However, they also have the following disadvantages: • They can be only used for DWH with «star» schema. • One query cannot benefit from both size reductions of intermediate tables with Bloom filter and duplication of small tables over the nodes during the join. T1 and T2 be linked with 1:M condition. Bloom filters are created for T1 table partitions. Then on collect command these filters are collected on the managing node and joined with the OR command. T2 table records are additionally filtered with the resulting filter.

$$(V/N/n)\cdot n\cdot N=V.$$

The number of fragments (n) equals Reduce task quantity.

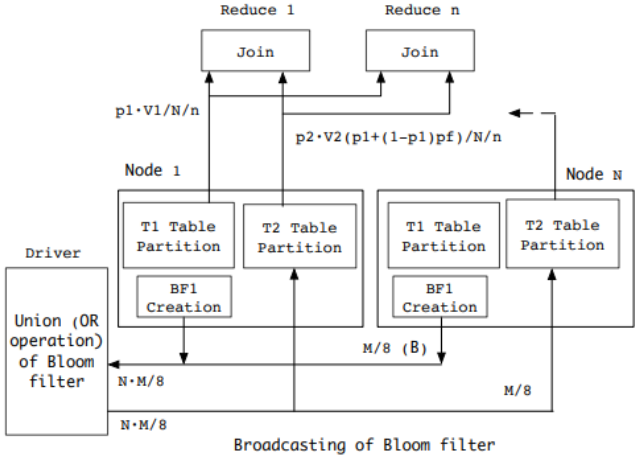
The second multiplier (n) represent the number of fragments per node. (N) indicates that fragments from all nodes are transmitted through the network.

The shuffle volume for T1 (see Fig. 1) filtered by the search condition (represented by p1 probability) equals:

$$V1\cdot p1.$$

The shuffle volume for a filtered table T2 (selection by subquery condition with probability p2 and after Bloom filter BF1 application) equals:

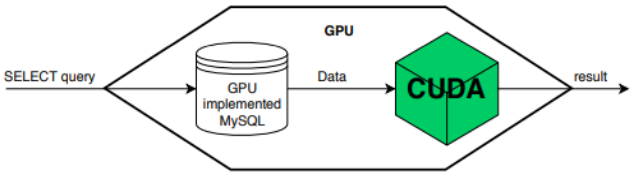
$$V2\cdot p2\cdot(p1\cdot 1+(1-p1)pf).$$



2.5 Efficiently running SQL queries on GPU

A simple system where the GPU's kernel reads a .csv file with data from the database table that it is needed to speed up the simple aggregation functions in the SQL query (Figure 1). This means that the system does not mimic the architecture of a relational database management system that is left for a future work. The proposed system architecture consists of a database implemented on the GPU and would use the GPU's memory for shorter data transfer latency and throughput. This system can be seen in Figure 2. We used MySQL as the foundation for the CPU counterpart. The MySQL database consists of a table with two attributes, 'id' and 'number', where the attribute 'id' is the primary key and is an auto-incremented value, and the 'number' attribute contains double values. The database table contains one million rows of real

Fig. 1. Data flow used for CUDA support of MySQL



(double) valued numbers within the range of [-200, 200], which is randomly generated with a coded procedure in the MySQL workbench. To get a feeling how MySQL compares with CUDA, SQL queries were executed 10 times, and an average time was calculated for each aggregation function, the following are the queries executed for the purpose of the project:

- SELECT SUM(number) from table
- SELECT MAX(number) from table
- SELECT MIN(number) from table

For each aggregation function kernel, we investigated what was the number of threads and the best grid/block combination for optimal performance. A program was written to generate the block size and grid size, considering that every block taken by the streaming multiprocessor runs in warps of 32 threads. This limited the size of the blocks to the numbers divisible by 32 with the maximum size of 1024 (hardware limitation), which is the grid size multiplied with the block size. The kernel that mimics the SUM aggregation function, is implemented as a one-dimensional problem, which means the block and grid size were one dimensional, and multidimensional implementation is left for future work. The MAX and MIN aggregation function kernels were implemented with two-dimensional block and grid sizes.

TABLE I MODERN ANALYTICAL DATABASES

	SLC	F-DML	F-ACID	GPD>M	SBSQL	CMHW
DB2		X	X	X	X	X
Netezza		X	X	X	X	
Teradata		X	X	X	X	
Greenplum		X	X	X	X	X
HANA		X	X		X	X
Hive	X			X		X
Spark	X				X	X
Spark	X				X	X

2.6 . Accelerating SQL with Complex Visual Querying

The OutSystems Platform provides a cloud solution of low code development, which allows developers to build and deploy enterprise-grade applications though visual interactions optimizing the time, effort, and previous knowledge necessary. Then, the OutSystems Platform aims to provide an application development environment that could be used by users with different backgrounds to build, deploy and manage their applications, using good practices and state-of-the-art technologies, even if they do not need to concern about that. Therefore, the vision and potential of the Out Systems Platform are similar to the Visual Querying Interface (VQIs) mentioned above since both intend to facilitate, accelerate, and optimize processes through visual interaction. The principal purpose of this visual interface is to provide a more visual and dynamic tool that accelerates the query formulation process and turns it easier and less error prone. So, the existing interface should be a useful and efficient tool for developers due to the potential of that visual approaches above-mentioned. However, OutSystems knew there were developers that were not using the visual querying interface, maintaining their preference for SQL. Therefore, it was necessary to verify the main causes that led users to not use that visual tool.

Paper Prototype: Simple low-fidelity prototype implemented in paper using a ruler, a set square, and writing materials. Through that approach, it is possible to implement the first ideas faster and reduce the risk of adoption failure, since the implemented ideas are already tested. The main concern of this type of prototype is that the design of the major interface changes might affect user's mental model which enables the early detection of which choices should continue to the next iterations or if they need to be redesigned. Service Studio Prototype: This is the final prototype, developed using C#, Type-script, and react, which are integrated in the new Design of the Service Studio. Through this prototype, the solutions implemented are validated and compared with the previous existing implementation in order to validate if the usability of the system has improved with the implemented changes.

2.7 Indexing for Large Scale Data Querying based on Spark SQL

In recent years, big data applications have gained more and more popularity both in academic communities and in business solutions. With the development of big data technologies, many frameworks which process big data emerge. This allows users to deal with TB or even PB of data conveniently. Apache Spark[2, 5] is a fast and general engine for large-scale data processing which provides high-level and expressive APIs in Java, Scala, Python and R. To achieve better performance, Spark keeps intermediate data in memory and optimizes physical execution. A series of built-in libraries including Spark SQL, MLlib, Spark Streaming and GraphX are incorporated into Spark to provide generality and scalability to build parallel applications. Spark SQL[3, 6] is a built-in component which enables programmers to perform relational operation through DataFrame API. Spark SQL is not designed for long-run services. In this paper, a new optimized component for Spark SQL which is based on fined-grained data format and customized index support is proposed. This new design allows users to create/drop index structures on specific columns that would be utilized in multiple queries, which could significantly accelerate SQL queries. Shark[7], an interactive SQL engine on Hadoop system, modified Apache Hive system to execute queries on external data stored in Hive. Shark only supports for querying external data, not for data in Spark programs (within RDD). Secondly, Spark SQL incorporates an extensible optimizer called Catalyst. It provides high performance using some DBMS optimization techniques, and enables programmers to add optimization rules and external data sources easily. Once the application is shutdown or restart, the table data cached in memory will be eliminated. On top of Spark SQL, we wanted to construct an index layer to speed up Spark SQL queries. This index layer enables users to customize index for ad-hoc queries on certain column(s) to accelerate query on large scale data sets further.

A. Design of Data Format

we split a complete data table files into many row groups to store in a data (table) file. Each row group is divided into several columns. Each column in a single row group is called a fiber. Data files are distributed among all nodes in a cluster. Create index operation will be parallelized among worker nodes. we present a new data format called "Spinach" which is also a columnar storage format. The whole table is divided into several row groups (the row group size could be configured by user), and within each row group, row

entries are divided according to columns. Each column in a row group is called a data fiber. All these fibers in a table is organized sequentially in a data file. we extended Spark SQL DDL (data definition language) to support create/drop index on specific columns. Programmers can use the following statements to create and drop index. create sindex idx1 on tableName(columnNane) drop sindex idx1 on tableName(columnName).

B. B+ Tree Index Design

Spark SQL allows users to query and process structured data resides in "DataFrame". When a SQL query on DataFrame is triggered, it actually triggers a full table scan or skip table scan using some statistics info the traditional approach in Spark SQL is not suited for random access. When index creation statement is triggered, B+ tree index partitioned files will be created concurrently on all worker nodes where partitioned data files reside.

In data segment, we only save the cluster of row ids (global row ids in each data file/table), because that keeping the whole row entries will make the index file so enormous that causes I/O bottleneck. At the beginning of building index, row keys will be sorted in ascending/descending order along with its row ids in memory. Keeping index keys in order is to facilitate the efficient sequential access to row entries during B+ tree range search. . After the model is built, row keys in ascending/descending order along with its start offset of row ids will be filled in each tree node sequentially according to the in-memory hash map. . After the model is built, row keys in ascending/descending order along with its start offset of row ids will be filled in each tree node sequentially according to the in-memory hash map. According the post-order traversal, the last node that will be record is the rootnode.

C. Architecture

There are 3 layers including data layer, control layer, and index selection layer. Data layer contains some data fields of B+ tree structure, the representation of range interval as well as some classes for locating current position in B+ tree. Control layer contains range scanner that could be used to traverse the B+ tree to find the target key(s). On top of the above two layers is index selection layer. The function of this layer is twofold: check whether we could use index to speed up query and select the best index among all of the available indexes.

D. Supporting For Query Functionalities

We implemented a filter predicate optimizer of query conditions to push down query filter predicates. Query conditions may be complicated which intermix with "AND" and "OR" conditions. The goals of this optimizer is to simplify filter predicates of query conditions to its most compact form. If a composite index is available for an ad-hoc query, the query might only use the first few columns of composite index, then we build a key schema for the first few columns that composite index matches. Because schema is lightweight, building schema to compare keys is quite efficient instead of constructing new internal rows. As a result, the number of columns of key schema are dynamic for each input query, but the row keys in B+ tree are static and it is unnecessary to create temporary row keys.

E. Index Selection

For single column index, if the index attribute could match attribute in the query hash map, then this index is available. For multi-column index, all of the index columns are scanned from left to right to check whether it could be matched with the attributes in the query hash map using left-most prefix matching. After getting the available indexes, the next step is to select the best index used to query. , for one index, if the query statement matches with the highest proportion of entries of this index, the index is treated as best index. The overhead of such selection could be ignored. In future work, we would implement cost based rules to select between multiple indexes.

The B+ tree index structure for this ad-hoc query engine mainly focuses on improving query performance on ad-hoc queries. We make comparisons among parquet, ad-hoc query engine Spinach, and Spark SQL on several dimensions shown in Table 1. From the chart, it is clearly that other architecture and strategies took a bit longer time to execute range search than full table scan respectively, because they need to traverse each row entries and filter out the ones that do not meet the query conditions. through the evaluations, our ad-hoc query engine combines advantages of Parquet and Spark SQL in-memory on data cache and locality, moreover, it supports predicate pushdown to enhance query efficiency. Benchmarks show that the system has higher performance on processing large scale data compared with original Spark SQL and parquet, which is more acceptable for big data applications

	Data Locality	Data Cache	Query Optimization	Data Restore
Parquet	HDFS file locality	HDFS cache	Optimization with statistical info	Data resides in DFS
Spinach	HDFS file locality + in memory cache awareness	Spark executor in-process off-heap memory	Customized index to support predicate pushdown	Data resides in DFS
Spark SQL in memory	In memory cache awareness	On-heap memory	Optimization with statistical info	Data resides in memory

Table 1. Comparisons about some dimensions

3. Comparison of compared techniques.

3.1) Speed

Paper 4 and Paper 6

The developed BFCA method is more than 2 times faster and more than 10 times better on shuffle volume (as compared to Spark SQL)

Paper 1 and Paper 6

SQLgen is up to 78× faster than DIABLO and up to 25× faster than hand-written RDD-based programs, giving performance close to that of hand-written programs in Spark SQL.

But Spark SQL is a bit faster than SQLgen.

Paper 2 and paper 3

The Oracle I1g RDBMS efficiency performance improves by approximately absolute 50%.The experiment results show that DOE achieves more than 100x performance improvement with PostgreSQL when compared to Oracle 11g RDBMS

.

3.2) Usability

Paper 5 and Paper 6

GPU is has faster processing than complex visual querying. But complex visual querying is more user- friendly than GPU. In paper 6 they have developed an interface which is easy for users to use.

3.3) Overall Performance

Paper 7 and paper 1

We use indexing to get data from a vast database. The results are efficient by using indexing in spark SQL. It is faster than retrieving data using SQLgen.

3.4) Accuracy

Paper 3 and paper 7

Oracle 11g RDBMS works efficiently on structured data by using portioning techniques but we cannot use them in bulk unstructured data. For this spark SQL works efficiently and gives accurate results.

4. Conclusion

We have explained and compared various methods which are used to increase the efficiency of query processing. We have presented a framework that translates array-based loops to Spark SQL. We have designed a whole stack database offloading system, DOE. And, we proposed a flexible DOE programming platform DP2, and designed an efficient Conflux accelerator architecture. We have taken the physical database tuning techniques to improve the performance of a relational database. In our proposed scenario the partitioning is one of the tuning techniques that enhance the performance approximately 50% of the results with the default configuration. In graph databases, the joins are saved like a database object and take constant time and are very efficient. We have studied a new method for complex SQL queries and the implementation was developed in the Spark environment in a package mode. It is based on the cascade Bloom filter application to intermediary query tables. We have showed the potential of GPU's accelerating database systems by developing kernels for the SQL aggregation functions. We have seen That the low-code development paradigm has accelerated software development and extended this industry to people with different backgrounds, due to the usage of visual programming languages. Sharing the same vision, VQIs arose in order to facilitate the data querying process, turning it easier to learn, more efficient, and less error prone. We have seen a new design for ad-hoc query based on Spark SQL, as respects of its background, structure, functionalities, and analysis as well as evaluation, is presented. This ad-hoc query engine extends Spark SQL as a pluggable module on supporting customized B+ tree indexes and provides unified data source interfaces for Spark SQL users to process structured data. We have used compared all the seven methods for efficiency, accuracy and overall performance.

5. References

1)M. H. Noor and L. Fegaras, "Translation of Array-Based Loops to Spark SQL," *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 469-476, doi: 10.1109/BigData50022.2020.9378136.

2)W. Lu, Y. Chen, J. Wu, Y. Zhang, X. Li and G. Yan, "DOE: Database Offloading Engine for Accelerating SQL Processing," *2022 IEEE 38th International Conference on Data Engineering Workshops (ICDEW)*, 2022, pp. 129-134, doi: 10.1109/ICDEW55742.2022.00026.

3)W. Khan, C. Zhang, B. Luo, T. Kumar and E. Ahmed, "Robust Partitioning Scheme for Accelerating SQL Database," *2021 IEEE International Conference on Emergency Science and Information Technology (ICESIT)*, 2021, pp. 369-376, doi: 10.1109/ICESIT53460.2021.9696761.

4)A. Burdakov *et al*., "Bloom Filter Cascade Application to SQL Query Implementation on Spark," *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 187-192, doi: 10.1109/EMPDP.2019.8671557.

5)D. Dojchinovski, M. Gusev and V. Zdraveski, "Efficiently Running SQL Queries on GPU," *2018 26th Telecommunications Forum (TELFOR)*, 2018, pp. 1-4, doi: 10.1109/TELFOR.2018.8611821.

6)S. Jyothi and T. S. Rajeswari, "Accelerating SQL with Complex Visual Querying," *2022 Second International Conference on Artificial Intelligence and Smart Energy (ICAIS)*, 2022, pp. 1126-1131, doi: 10.1109/ICAIS53314.2022.9742759.

7) Y. Cui, G. Li, H. Cheng and D. Wang, "Indexing for Large Scale Data Querying Based on Spark SQL," 2017 IEEE 14th International Conference on e-Business Engineering (ICEBE), 2017, pp. 103-108, doi: 10.1109/ICEBE.2017