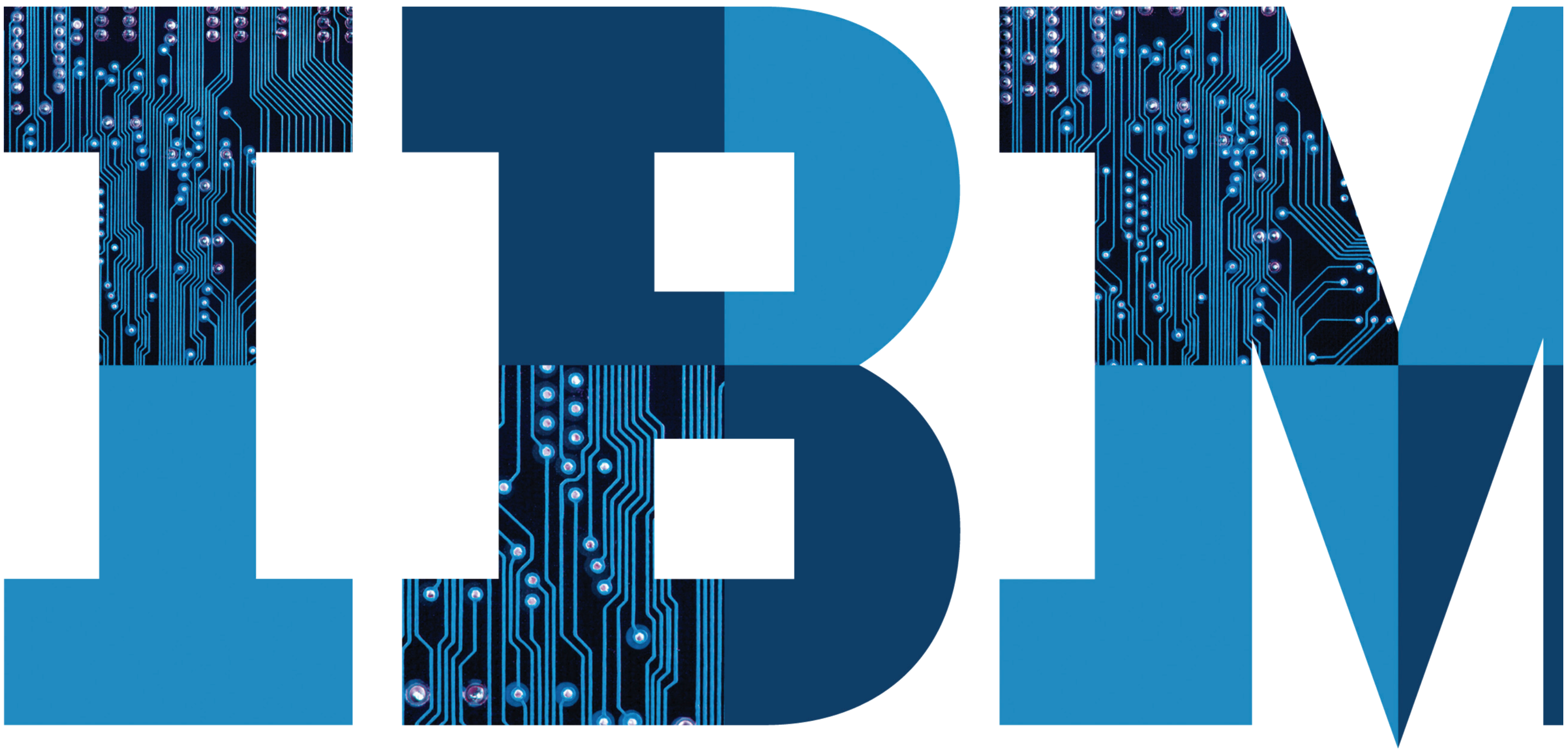
IBM Software Group
Thought Leadership White Paper

April 2014

SQL-on-Hadoop without compromise

How Big SQL 3.0 from IBM represents an important leap forward for speed, portability and robust functionality in SQL-on-Hadoop solutions

By Scott C. Gray, Fatma Ozcan, Hebert Pereyra,
Bert van der Linden and Adriana Zubiri



Contents

Introduction

When considering SQL-on-Hadoop, the most fundamental question is: What is the right tool for the job? For interactive queries that require a few seconds (or even milliseconds) of response time, MapReduce (MR) is the wrong choice. On the other hand, for queries that require massive scale and runtime fault tolerance, an MR framework works well. MR was built for large-scale processing on big data, viewed mostly as "batch" processing.

As enterprises start using Apache Hadoop as a central data repository for all data — originating from sources as varied as operational systems, sensors, smart devices, metadata and internal applications — SQL processing becomes an optimal choice. A fundamental reason is that most enterprise data management and analytical tools rely on SQL.

As a tool for interactive query execution, SQL processing (of relational data) benefits from decades of research, usage experience and optimizations. Clearly, the SQL skills pool far exceeds that of MR developers and data scientists. As a general-purpose processing framework, MR may still be appropriate for ad hoc analytics, but that is as far as it can go with current technology.

The first version of Big SQL from IBM (an SQL interface to IBM InfoSphere® BigInsights™ software, which is a Hadoop-based platform) took an SQL query sent to Hadoop and decomposed it into a series of MR jobs to be processed by the cluster. For smaller, interactive queries, a built-in optimizer rewrote the query as a local job to help minimize latencies. Big SQL benefited from Hadoop's dynamic scheduling and fault tolerance. Big SQL supported the ANSI 2011 SQL standard and introduced Java Database Connectivity (JDBC) and Open Database Connectivity (ODBC) client drivers.

Big SQL 3.0 from IBM represents an important leap forward. It replaces MR with a massively parallel processing (MPP) SQL engine. The MPP engine deploys directly on the physical Hadoop Distributed File System (HDFS) cluster. A fundamental difference from other MPP offerings on Hadoop is that this engine actually pushes processing down to the same nodes that hold the data. Because it natively operates in a shared-nothing environment, it does not suffer from limitations common to shared-disk architectures (e.g., poor scalability and networking caused by the need to move "shared" data around).

Big SQL 3.0 introduces a "beyond MR" low-latency parallel execution infrastructure that is able to access Hadoop data natively for reading and writing. It extends SQL:2011 language support with broad relational data type support, including support for stored procedures. Its focus on comprehensive SQL support translates into industry-leading application transparency and portability. It is designed for concurrency with automatic memory management and comes equipped with a rich set of workload management tools. Other features include scale out parallelism to hundreds of data processing nodes and scale up parallelism to dozens of cores. With respect to security, it introduces capabilities on par with those of traditional relational data warehouses. In addition, it can access and join with data originating from heterogeneous federated sources.

Right now, users have an excellent opportunity to jump into the world of big data and Hadoop with the introduction of Big SQL 3.0. In terms of ease of adoption and transition for existing analytic workloads, it delivers uniquely powerful capabilities.

1. Architecture

Big SQL 3.0 is an SQL-on-Hadoop solution that moves away from traditional MR frameworks. It introduces a shared-nothing parallel database architecture that leverages state-of-the-art relational database technology from IBM. A key differentiator from other SQL-like systems over Hadoop is that the data remains on the HDFS cluster. Big SQL 3.0 does not impose any relational database management system (RDBMS) structure or restrictions on the layout and organization of the data, which helps maintain the flexibility of Hadoop. The database infrastructure provides a logical view of the data (by allowing storage and management of metadata) and a view of the query compilation, plus the optimization and runtime environment for optimal SQL processing. Big SQL 3.0 exploits sophisticated query rewrite transformations targeted at complex nested decision support queries.

Parallel processing

Figure 1 illustrates the general architecture of Big SQL 3.0. A user configures the server so that user applications connect on a specific node. This is where SQL statements will be routed. Commonly referred to as the coordinating node, this is where SQL statements will be compiled and optimized to generate a parallel execution query plan. A runtime engine then distributes the parallel plan to worker nodes and orchestrates the consumption and return of the result set. Big SQL 3.0 does not require a worker node on every HDFS data node. It can operate just as efficiently in a configuration where it is deployed on a subset of the Hadoop cluster.

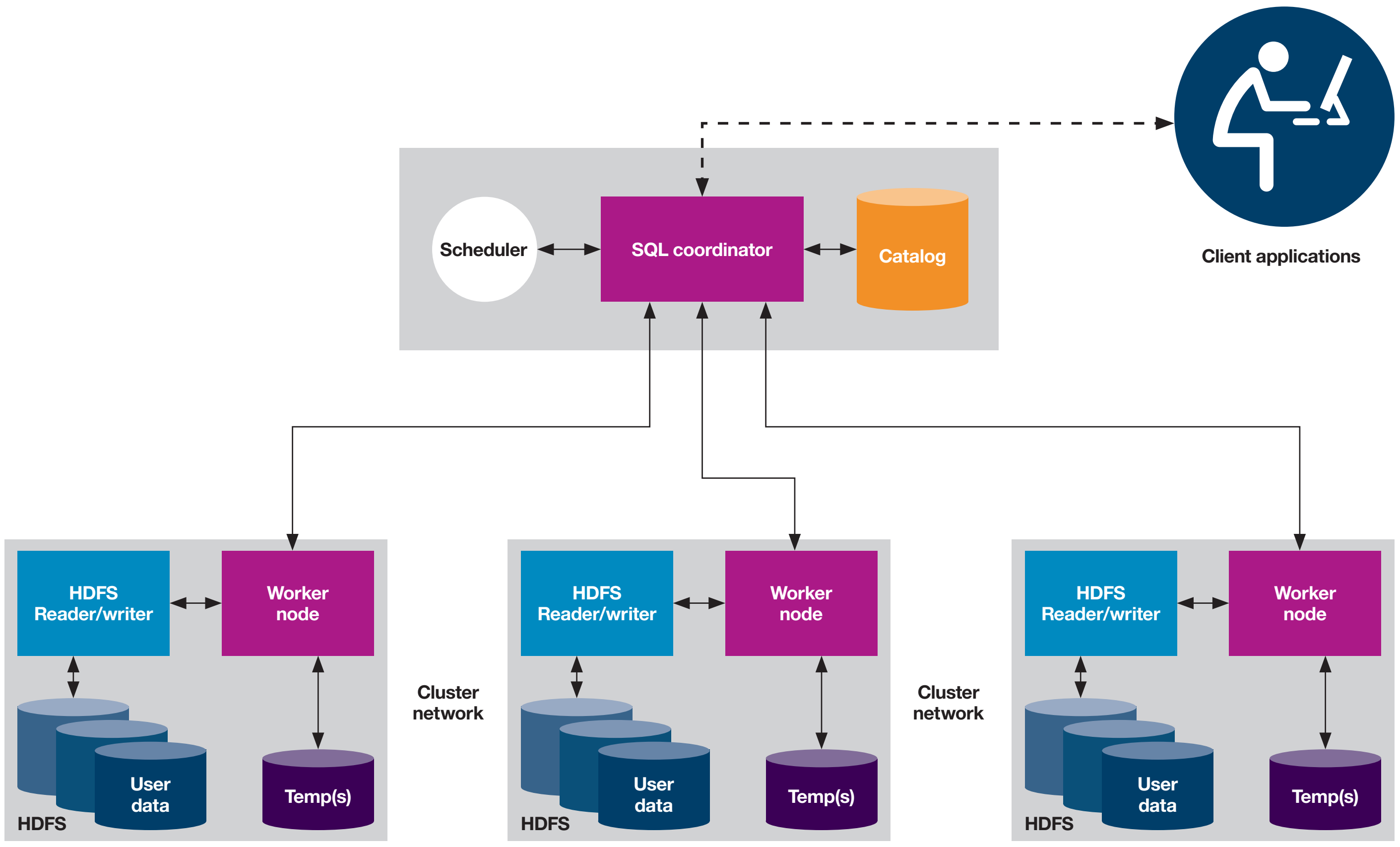


Figure 1: Big SQL 3.0 architecture

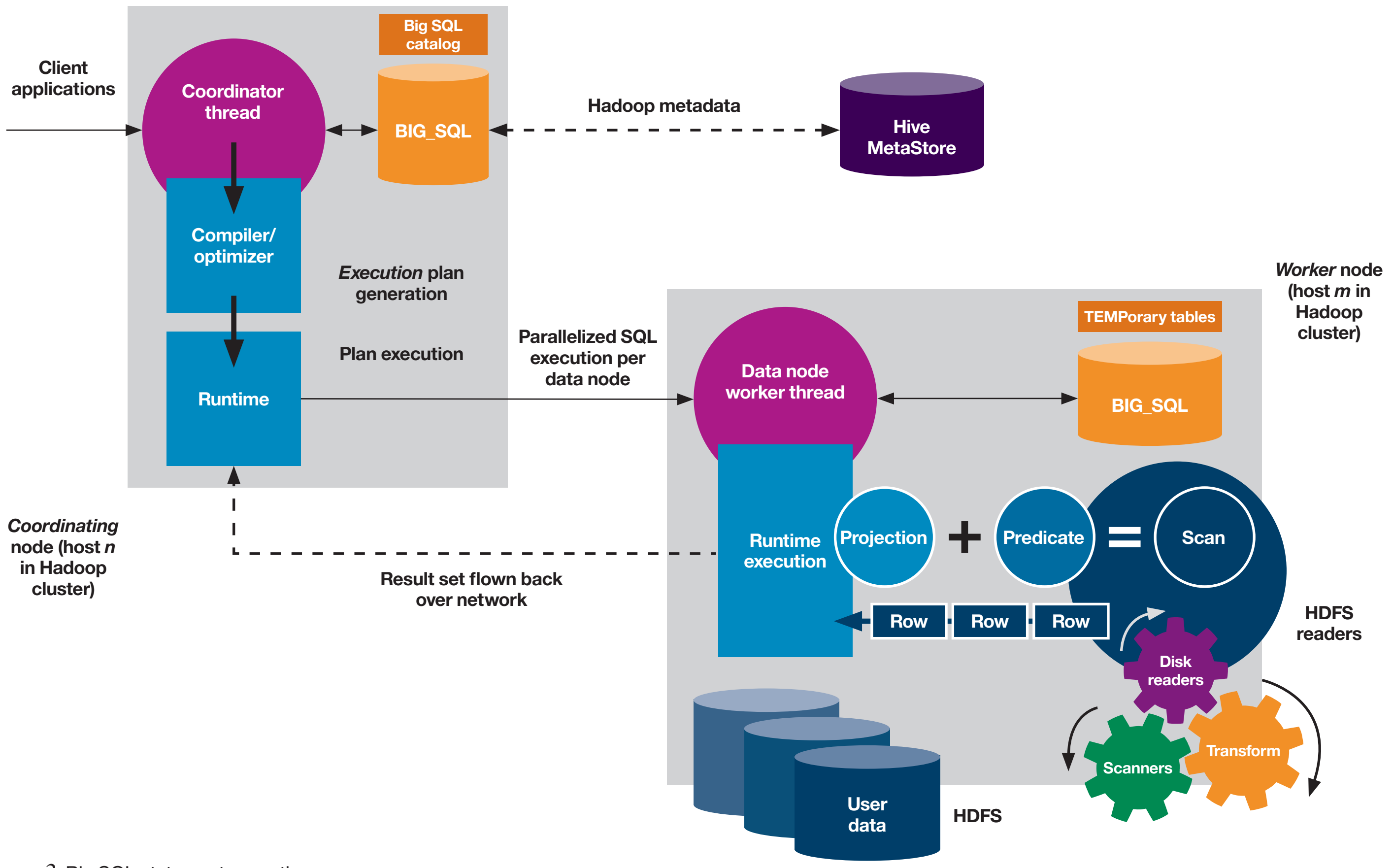


Figure 2: Big SQL statement execution

Once a worker node receives a query plan, it dispatches special processes that know how to read and write HDFS data natively. This is illustrated in Figure 2. Big SQL 3.0 utilizes native and Java open source–based readers (and writers) that are able to ingest different file formats. To improve performance, the Big SQL engine pushes predicates down to these processes so that they can, in turn, apply projection and selection closer to the data. Another key task performed by these processes is the transformation of input data into an appropriate format for consumption inside Big SQL.

This transformation allows for intermediate data to be loaded into the Big SQL engine memory for additional processing. Big SQL 3.0 utilizes a variety of in-memory buffer pools to help optimize query and utility-driven processing.

In addition to distributed MPP processing, each worker node further parallelizes execution both inside the Big SQL engine and on the HDFS side. Big SQL 3.0 automatically determines the best degree of intra-query parallelism dynamically, per query, based on available resources.

Metadata

Big SQL 3.0 utilizes the Apache Hive database catalog for table definitions, location, storage format and encoding of input files. This means it is not restricted to tables created and/or loaded via Big SQL. As long as the data is defined in the Hive Metastore and accessible in the Hadoop cluster, Big SQL can get to it.

Some of the metadata from the Hive catalog is stored locally by Big SQL for ease of access and to facilitate query execution. The Big SQL catalog resides on the headnode. This allows for local access during query compilation and optimization. Query plans for stored procedures are stored as packages in the catalog for future re-execution or execution by other client applications (which helps save the recompile cost). These packages are managed automatically and invalidated, for example, whenever a topology change is detected in the Big SQL cluster.

Scheduler

A fundamental component in Big SQL 3.0 is the scheduler service. This scheduler acts as the Big SQL liaison between the worlds of SQL and Hadoop. It provides a number of important services:

Hive Metastore interface

When queries are compiled and optimized, the Big SQL catalog is queried for basic information, such as column names and data types. Further optimizations can be achieved by understanding the physical layout of the data and associated statistics. The Big SQL engine will contact the scheduler at various stages of execution to push down and retrieve critical metadata.

Partition elimination

Big SQL 3.0 supports Hive partitioned tables, so the compiler will push predicates down to the scheduler to eliminate partitions that are not relevant for a given query.

Scheduling of work

The scheduler maintains information about where and how data is stored on Hadoop. Big SQL uses that information to help ensure that work is processed efficiently, as close to the data as possible. In traditional shared-nothing architectures, one of the tasks of the compiler-optimizer is deciding which nodes to involve in a query. This is usually achieved through control of data partitioning. Big SQL 3.0, however, does not impose any database-style partitioning. The system instead supports the concept of scattered (or dynamic) partitioning, in which no assumptions are made on what nodes the data should be accessed from. This allows Big SQL not only to assign work where the data resides locally but also to take into account worker node load and other hints for query plan distribution.

Concurrency and workload management

Enterprise solutions over Hadoop are limited to experimental propositions unless they can handle real-life concurrent business intelligence (BI) and analytics workloads. This is where Big SQL 3.0 offers a novel architecture by coupling the flexibility of HDFS with mature, multi-tenant and proven data warehouse technology from IBM.

The problem of memory management has been pervasive in Hadoop ecosystems in part because it is less mature than other frameworks. Much of the tuning exercise in Hadoop becomes an intensive trial and error process: reacting to out-of-memory exceptions, adjusting service properties, rerunning queries. It is a very workload-specific exercise, one that requires adjustment as the workload changes: one workload might need more mapper memory while another might have higher reducer memory requirements.

Big SQL 3.0 introduces automatic memory tuning. At installation, the user specifies how much memory is available for the system to use, and Big SQL optimizes execution within these limits. There are two key aspects to this fully autonomic behavior: (1) use memory efficiently and (2) prevent out-of-memory exceptions. This is particularly important when the workload involves multiple concurrently running applications.

Big SQL 3.0 manages concurrent workload automatically by balancing resources across executing queries. Memory resources are distributed across shared buffer pools that light threads tap into as needed.

The Big SQL system is designed to constantly modify memory configuration to maximize overall system performance. By monitoring internal metrics on resource consumption, it can react to changing conditions. These same metrics, combined with built-in tuning algorithms, allow it to make predictive decisions and tune the system ahead of sizable changes in demand.

The system utilizes a self-tuning memory management infrastructure that dynamically and automatically tunes memory areas based on the size of the physical host the worker node (data partition) is running on. As a result, cluster configuration does not need to be homogeneous, with each worker node managed and tuned independently for performance and concurrency.

In addition, Big SQL 3.0 introduces a comprehensive set of workload management tools that provide admission control for various classes of concurrent queries. The user, for example, can configure the system by defining a workload management class to limit the number of concurrent "heavyweight" analytical SQL requests (subsequent heavyweight queries are queued until an existing one completes). In some environments, this can help avoid over-stressing CPU and memory resources, as well as disk I/O, and enable Big SQL to coexist amicably with other system consumers within the same Hadoop cluster.

Fault tolerance and cluster elasticity

Big SQL 3.0 automatically maintains the status of worker nodes. This information, coupled with the scheduler-driven scattered partitioning, allows the system to distribute query execution to only those nodes that are up and running. If one or more nodes crash, or the user wants to bring nodes down for maintenance, the Big SQL engine will use that information to automatically (and transparently) redistribute new queries.

The same applies to changes in the cluster topology. The user may add or remove worker nodes online, which will be immediately detected by the scheduler and reflected in its decisions.

2. Rich SQL

For many, if not most SQL-on-Hadoop technologies, the primary focus appears to be either sheer performance (at the expense of SQL expressiveness) or breadth of data sources (at the expense of SQL expressiveness as well as performance). Big SQL 3.0 brings together a modern, mature IBM SQL compiler and runtime engine working hand-in-hand with the Hive database catalog and I/O layers to allow expressive SQL2011 queries that execute with enterprise-class levels of performance for the same table definitions and the same flexible interfaces that are shared across the Hadoop ecosystem.

Big SQL in IBM InfoSphere BigInsights 2.1.0 introduced industry-leading SQL query capabilities. With Big SQL 3.0, these capabilities have been extended even further with the introduction of significant levels of SQL PL compatibility, including stored procedures, SQL-bodied functions and a rich library of scalar, table and online analytical processing (OLAP) functions. By aligning its SQL capabilities with the established standards of SQL PL, Big SQL 3.0 introduces a new level of SQL compatibility between database platforms and dramatically opens up interoperability with established BI tools.

Queries

Big SQL 3.0 continues to establish Big SQL as one of the richest SQL query dialects over data in Hadoop, allowing users to express the queries in a familiar version of SQL rather than a limited subset or modified version of the language. This section highlights some of the features that set Big SQL apart in this regard.

Actual BIG queries

The Big SQL runtime engine provides in-memory caching capabilities as well as the ability to spill large data sets to the local disk at each node that is processing a query, meaning that the only limit to the size of the queries, groups and sorting is the disk capacity of the cluster.

Subqueries

In SQL, subqueries are a way of life. They allow for much more flexibility in expressing the query, improved readability and, in some cases, improved performance. Yet many SQL-on-Hadoop solutions have significant limitations for when and where subqueries may be used (if they can be used at all) and how deeply they may be nested.

Big SQL 3.0 allows subqueries to be used anywhere that an expression may be used, such as in a SELECT list:

```
SELECT FNAME, LNAME,
  (SELECT NAME
    FROM DEPARTMENT AS DEP
  WHERE DEP.DEPT_ID = EMP.DEPT_ID)
FROM EMPLOYEE AS EMP
```

Subqueries may be correlated (as shown above) and uncorrelated.

```
SELECT *
  FROM EMP_ACT
 WHERE EMPNO IN
(SELECT EMPNO
  FROM EMPLOYEE
 WHERE WORKDEPT = 'E11')
```

They may be used in any clause that the SQL standard allows for subqueries:

```
SELECT WORKDEPT, MAX(SALARY)
  FROM EMPLOYEE EMP_COR
 GROUP BY WORKDEPT
HAVING MAX(SALARY) <
  (SELECT AVG(SALARY)
    FROM EMPLOYEE
  WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

The Big SQL query rewrite engine aggressively decorrelates subqueries whenever possible, taking into account additional table metadata, such as nullability and constraint information, to help ensure the most high-performing query execution possible.

Table expressions

Beyond the ability to simply query tables, Big SQL provides a number of table expressions that may be used as sources of data for queries. This includes common table expressions:

```
WITH
  PAYLEVEL
  AS
    (SELECT EMPNO, YEAR(HIREDATE) AS HIREYEAR,
           EDLEVEL, SALARY+BONUS+COMM AS TOTAL_PAY
      FROM EMPLOYEE
     WHERE EDLEVEL > 16),
  PAYBYED (EDUC_LEVEL, YEAR_OF_HIRE, AVG_TOTAL_PAY)
  AS
    (SELECT EDLEVEL, HIREYEAR, AVG(TOTAL_PAY)
      FROM PAYLEVEL
    GROUP BY EDLEVEL, HIREYEAR)
SELECT EMPNO, EDLEVEL, YEAR_OF_HIRE, TOTAL_PAY,
      AVG_TOTAL_PAY
  FROM PAYLEVEL, PAYBYED
 WHERE EDLEVEL = EDUC_LEVEL
   AND HIREYEAR = YEAR_OF_HIRE
   AND TOTAL_PAY < AVG_TOTAL_PAY
```

Built-in or user-defined table functions also may be used as data sources, potentially allowing queries to interact with external data sources:

```
SELECT FNAME, LNAME, EMP_IMAGES.IMAGE
  FROM EMPLOYEE AS EMP,
       TABLE(getImage(
         'http://www.myco.com/emp/images',
         EMP.EMP_ID)) AS EMP_IMAGES
 WHERE EMP.FNAME = 'Fred'
```

In addition, static data may be provided using a VALUES expression:

```
SELECT EMP.EMP_ID, BRACKETS.BRACKET
  FROM EMPLOYEE AS EMP,
       TABLE(VALUES (1, 0, 10),
                    (2, 11, 18),
                    (3, 19, 999))
           AS BRACKETS (BRACKET, MIN, MAX)
 WHERE EMP.AGE >= BRACKETS.MIN
   AND EMP.AGE <= BRACKETS.MAX
```

In addition to standard derived queries, Big SQL provides for full lateral join support (beyond just lateral table function and collection provided by some other solutions):

```
SELECT P.PNO, P.PNAME, L.SUM_QTY
  FROM PARTS P,
       LATERAL(
          SELECT SUM(QTY) SUM_QTY
            FROM SUPPLY SP
           WHERE SP.PNO = P.PNO) AS L
```

Joins

Big SQL supports the full contingent of standard join operations, from the standard implicit SQL join syntax:

```
  FROM T1, T2, T3
 WHERE T1.C1 = T2.C2 AND T3.C4 = T2.C3
```

To the explicit ANSI join syntax:

```
  FROM T1 INNER JOIN T2 ON T1.C1 = T2.C2
   LEFT OUTER JOIN T3 ON T3.C4 = T2.C3
```

In all cases, the Big SQL query optimizer will ensure optimal join order based on available table statistics and heuristics.

Most other SQL-on-Hadoop solutions restrict join operations to only equality join operations and will not support CROSS JOIN (Cartesian products) such as:

```
FROM T1, T2
WHERE T1.C1 = 10 AND T2.C3 = 'Bob'
```

Or:

```
FROM T1 CROSS JOIN T2
WHERE T1.C1 = 10 AND T2.C3 = 'Bob'
```

Or queries involving non-equality joins, such as:

```
FROM T1, T2
WHERE T1.C1 > T2.C1
```

These restrictions exist because such join operations are difficult to implement efficiently on such a system as Hadoop, in which data is randomly scattered across the nodes of the cluster and true indexes are generally not present. However, there are certainly classes of queries in which one of the two tables being joined is small or the filtering criteria on the table results in a small data set, in which case these queries can be effectively executed.

Big SQL provides the ability to perform all valid SQL standard join operations.

Join strategies

Beyond the ability to simply express joins using all of the available SQL standard join constructs, Big SQL provides for a number of strategies for implementing joins, automatically selecting the most appropriate join for a query based on available table statistics. These strategies include hash join, merge join and nested loop join.

Big SQL implements a parallel hybrid hash join algorithm that is designed to be optimized for clusters of highly symmetric multiprocessor (SMP) systems and can adapt its behavior to the resources available in the system. Its memory-conscious algorithm dynamically balances the workload when processing buckets of data stored in a disk. Buckets are scheduled depending on the amount of memory available, and the processes assigned to each bucket are dynamically chosen. This results in an efficient use of the shared memory pool, which saves I/O by avoiding hash loops, and helps minimize memory contention with a smart balancing of processes across buckets in the memory pool.

Windowing and analytic functions

Hadoop is a system designed for large-scale analytics. One of the key SQL features created to address analytic problems are SQL OLAP functions. Big SQL supports all of the standard OLAP specifiers, including the following:

- LEAD
- LAG
- RANK
- DENSE_RANK
- ROW_NUMBER
- FIRST_VALUE
- LAST_VALUE
- RATIO_TO_REPORT
- Standard aggregate functions

With the following analytic aggregates and functions:

- CORRELATION
- COVARIANCE
- STDDEV
- VARIANCE
- REGR_AVGX
- REGR_AVGY
- REGR_COUNT
- REGR_INTERCEPT
- REGR_ICPT
- REGR_R2
- REGR_SLOPE
- REGR_SXX
- REGR_SXY
- REGR_SYY
- WIDTH_BUCKET
- VAR_SAMP
- VAR_POP
- STDDEV_POP
- STDDEV_SAMP
- COVAR_SAMP
- COVAR_POP

Additional query features

The following is a list of additional significant features provided by Big SQL 3.0.

- **Grouping sets, cube and rollup** – A grouping set effectively allows for multiple GROUP BY expressions to be performed on a query, with CUBE and ROLLUP operations indicating how to aggregate data within subgroups. For example:

```
SELECT COALESCE(R1,R2) AS GROUP,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
  FROM SALES,
       (VALUES(‘GROUP 1’,’GROUP 2’)) AS X(R1,R2)
 GROUP BY GROUPING SETS (
     (R1, ROLLUP(WEEK(SALES_DATE),
        DAYOFWEEK(SALES_DATE))),
     (R2,ROLLUP( MONTH(SALES_DATE), REGION ) ) )
 ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION
```

- **Union operations** – Big SQL provides the full contingent of standard union operations, such as UNION [ALL], EXCEPT [ALL] and INTERSECT [ALL].

Inserting data

The standard SQL INSERT statement provides the ability to append new data to existing tables. The INSERT operation is supported for all available storage types and provides for adding data via the VALUES clause:

```
INSERT INTO T1 (C1, C2, C3)
VALUES (1, ‘Fred’, 3.4), (2, ‘Bob’, 19.2)
```

Or adding rows via a separate query:

```
INSERT INTO T1
SELECT C4, C3, C8 / 8.2 FROM T2 WHERE C9 > 100
```

The VALUES form of INSERT is recommended only for testing because it cannot be parallelized and because each call to INSERT…VALUES produces a single underlying data file.

When combined with a table UDF, an INSERT INTO… SELECT may be used to populate a table from external data sources that may not have been otherwise utilized via a normal SELECT or LOAD operation:

```
INSERT INTO T1
SELECT *
FROM TABLE(
   myHttpGet(
     ‘http://www.mydata.co/data?id=10’))
   AS REMOTE
WHERE REMOTE.CALLID > 20
```

Routines

Routines are application logic that is managed and executed within the Big SQL database manager, allowing for encapsulation and execution of application logic at the database rather than within the application. Beyond simply centralizing the application logic within the database, routines also allow for fine-grained access control. Database users can be granted permission to execute routines that access data sources that the user would not otherwise be capable of accessing. Additional application-specific security checks may be implemented within the routines in addition to the normal SQL security enforcement.

Big SQL supports the following types of routines:

- **Scalar functions** – A function that takes zero or more arguments and returns a single value. This value may be of any SQL data type.
- **Table functions** – A function that takes zero or more arguments and returns rows of values. Table functions may be used in the FROM clause of SELECT statements.
- **Procedures** – A procedure encapsulates application logic that may be invoked like a programming subroutine, including both input and output parameters. Procedures may include flow-of-control logic and error handling and may invoke static or dynamic SQL statements.

Routine development

Routines may be developed in a number of popular programming languages:

- **SQL** – Routines may be developed and managed entirely in SQL PL. Being a high-level language, SQL PL allows for rapid application development with very good performance because the routines are executed entirely within the Big SQL runtime engine.
- **Java** – Java routines provide all of the flexibility of the Java programming language, along with access to the enormous wealth of available Java libraries. Java routines are ideal for interacting with external data sources or accessing complex functionality that would not otherwise be accessible via SQL routines. In addition, the Java virtual machine (JVM) sandbox provides increased reliability and security over routines implemented in C/C++.
- **C/C++** – C/C++ routines allow for very low-level, efficient execution and direct access to native development libraries. These routines provide a very high level of execution performance but do not protect the developer from memory and pointer manipulation mistakes.
- **Mixture** – A routine developed in one language may invoke a routine in another language. For example, an SQL-bodied stored procedure may execute a SELECT statement in which a C++ function is applied to a column in the result set of the query.

Routines and security

Beyond the basic ability to grant individual users, groups or roles the ability to manage or execute routines, the privileges of the routine itself may be finely tuned. Specifically:

- Static SQL within the routines execute with the privileges of the user that defined the routine.
- Dynamic SQL within the routines is invoked under the privileges of the user invoking the routine, by default, restricting the risk of accidental privilege escalation, such as SQL injection attacks. If desired, this behavior may be modified to allow for execution of dynamic SQL under the rights of the package or routine owner.

Interoperability of routines

For all languages in which routines may be developed (SQL PL, Java or C/C++), Big SQL shares most common language features and functionality with IBM DB2® for Linux, UNIX and Windows and IBM DB2 for z/OS® database software, meaning that the same routine developed for Big SQL can be easily migrated between systems with a few (or no) modifications.

In addition, Big SQL provides standards-compliant APIs. By adhering to the standard APIs, such as SQLJ routines for Java and PARAMETER STYLE SQL or PARAMETER STYLE GENERAL routines for C/C++, routines that are developed for Big SQL can be shared across any other database platform implementing the standard.

Data ingestion

In addition to the existing Hadoop mechanisms for moving data, Big SQL provides a LOAD statement that is more familiar to the SQL user. Users can load flat-file data into Big SQL, and the SQL LOAD HADOOP statement allows a table to be populated from external data sources via standard JDBC drivers:

```
LOAD HADOOP
    USING JDBC CONNECTION
      URL 'jdbc:db2://myhost:50000/SAMPLE'
      WITH PARAMETERS (
        user = 'myuser',
        password='mypassword')
    FROM SQL QUERY
      'SELECT * FROM STAFF
        WHERE YEARS > 5 AND $CONDITIONS'
  SPLIT COLUMN ID
  INTO TABLE STAFF_Q
  APPEND
```

The data loading process is coordinated in parallel, with multiple nodes of the cluster processing ranges of values from the source system. This allows for parallelism within the Hadoop cluster as well as the ability to take advantage of any partitioning capabilities of the source from which data is being extracted.

Other features

The following are additional features available in Big SQL 3.0.

- **Built-in functions** – Big SQL 3.0 comes with an extensive library of more than 250 built-in functions and 25 aggregates.
- **Additional SQL types** – Big SQL tables share infrastructure with Hive and, therefore, are limited to the data types available within Hive (or slight extensions of those types). However, the SQL runtime in Big SQL supports a number of additional types that may be utilized in queries and routines if desired. These data types include: CLOB, BLOB, GRAPHIC, TIME, variable length TIMESTAMP(n), CURSOR, XML, associative arrays and user-defined types.
- **Global (session) variables** – Big SQL provides standard SQL PL syntax for defining global or session variables. These variables may be referenced in queries, stored procedures or functions. For example:

```
CREATE VARIABLE COMPANY_NAME
  VARCHAR(100) DEFAULT 'MyCo, Inc.'
  SELECT COUNT(*)
  FROM SALES AS S
 WHERE S.COMPANY_NAME = COMPANY_NAME
```

- **Duration arithmetic** – While a comprehensive set of scalar functions is provided for manipulating TIMESTAMP values, Big SQL provides SQL standard duration arithmetic operations to simplify the readability and maintainability of your SQL:

```
SELECT COUNT(*)
  FROM SALES
 WHERE SALE_DATE <
  CURRENT TIMESTAMP - 30 DAYS
```

- **Temporary tables** – During query execution, Big SQL may maintain one or more internal temporary tables. These tables are stored and managed in a highly efficient internal format and cached in memory, if possible, or spilled onto the local disk at each node as necessary.

3. Application portability and integration

The richness, breadth and depth of SQL provided by Big SQL 3.0 is about much more than enabling the ability to build robust database applications. It is equally about not having to build these applications from scratch. By supporting a vast range of SQL standard constructs — beyond just the ability to SELECT data — Big SQL creates an opportunity to reuse and share application logic among database platforms and to leverage the knowledge base of application developers on a completely new platform, such as Hadoop.

Open data

SQL is a very useful language. Thanks to standardization and decades of maturation, this well-known language has proven adept at solving many significant problems. It is not, however, the only language available. Nor is it always the best solution for every problem. Hadoop, in particular, has an ever-expanding array of languages and tools for analyzing massive quantities of data.

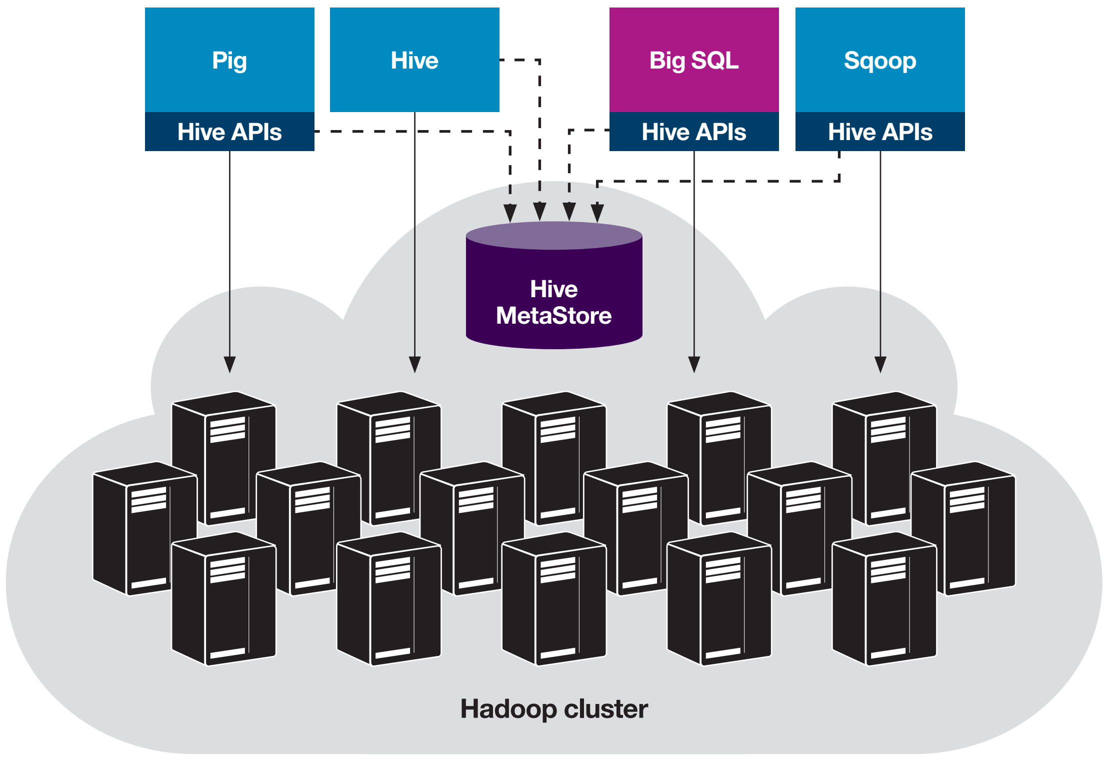


Figure 3: Hadoop cluster

Big SQL is a harmonious member of this ecosystem. All data living within Hadoop remains in its original format. There is no propriety storage format for Big SQL on the Hadoop file system. Hadoop table definitions are defined within and shared with the Hive Metastore, the de facto standard catalog for Hadoop. Similarly, Big SQL shares the same Hive interfaces for physically reading, writing and interpreting the stored data.

All of this means that there is no data store "in" Big SQL; the data is out there and open for access via the most appropriate tool available. A table defined in Big SQL is a table defined in Hive and vice versa. Thanks to projects like Apache HCatalog, a table defined in Hive is a data source available for all tools. Ultimately, these are simply files on a massive, shared, distributed file system that are accessible via any Hadoop application.

Common client drivers

Beyond the ability to share SQL across platforms, Big SQL 3.0 utilizes IBM Data Server client drivers, allowing the same standards-compliant JDBC, JCC, ODBC, CLI and .NET drivers to be used across Big SQL; DB2 for Linux, UNIX and Windows; DB2 for z/OS; and IBM Informix® database software.

Sharing the same driver across these platforms means that other languages that already leverage these drivers (such as Ruby, Perl, Python and PHP) may immediately begin to interact with Big SQL with no additional custom configuration or drivers.

Combining the ability to share much of the SQL dialect across IBM platforms and client drivers with the same behavior across platforms dramatically widens the number of applications capable of transparently interacting between the traditional database management system (DBMS) or data warehouse and Big SQL.

Enabling business analytics

BI tools, such as IBM Cognos® software, Microstrategy and Tableau, allow business users to visually express complex queries, as well as think and work in terms of business concepts, without necessarily having to understand the intricacies of the relational database and SQL. Once users have described what they want and how it should be visualized, it is the duty of these tools to determine how best to implement the user's request against the back-end database.

There are two important aspects of using BI tools:

- They are aware of the capabilities of the database they are querying and will attempt to express the user's request in the SQL dialect of the target platform (for platforms that speak SQL), using the available capabilities of that platform the best it can. For anything the target platform cannot express, these tools compensate by implementing the missing features themselves. In this way, these tools can query a Microsoft Excel spreadsheet just as well as they can query a full-blown RDBMS.
- In general, they are not query optimizers. They will push down as much work as possible to the target platforms and will attempt to express the query in an efficient manner, but they generally have no statistics to determine the best way in which to implement the query.

What this means is that for platforms with a limited SQL dialect, much of the query processing logic may be implemented by transferring massive quantities of data back to the BI application. For platforms with little or no optimization capabilities, it may be uncertain whether the portions will be processed efficiently.

The expressiveness of the SQL dialect used by Big SQL ensures that these tools will attempt to run as much of the processing as possible through Big SQL. Its mature query rewrite and optimization engines will take care of executing the query in the most efficient manner possible.

4. Enterprise capabilities

The pace of innovation and change in the Hadoop and analytics marketplaces is extremely fast. Many providers are adding SQL capability into their distribution or greatly enhancing Hive. However, in order to support the business, having an SQL interface is not enough. As more critical applications are deployed in these environments, it is important that the enterprise level capabilities provided for Hadoop are at the same level as those provided for the relational world. Governance, security and privacy issues need to be resolved. In addition, as customers start building true hybrid data warehouses, there is a need to join and query data no matter where it resides. Big SQL 3.0 provides these critical capabilities.

Security and auditing

There are two modes of security access control to Big SQL data and functions. Access to Big SQL is managed by facilities that reside outside the Big SQL server (authentication), while access within the Big SQL server is managed by Big SQL itself (authorization).

Authentication

Authentication is the process by which a system verifies a user's identity. User authentication is completed by a security facility outside Big SQL, through an authentication security plug-in module. A default authentication security plug-in module that relies on operating system–based authentication is included when Big SQL is installed. Big SQL also ships with authentication plug-in modules for Kerberos and Lightweight Directory Access Protocol (LDAP). For even greater flexibility in accommodating specific authentication needs, users can build a custom authentication security plug-in module. The authentication process produces a Big SQL authorization ID. Group membership information for the user is acquired during authentication.

Authorization

After a user is authenticated, Big SQL determines if that user is allowed to access data or resources. Authorization is the process by which Big SQL obtains information about the authenticated user, indicating which operations that user can perform and which data objects that user can access. When an authenticated user tries to access data, these recorded permissions are compared with the following permissions:

- The authorization name of the user
- The groups to which the user belongs
- The roles granted to the user directly or indirectly through a group or a role

Based on this comparison, the Big SQL server determines whether to allow the requested access. A privilege defines a single permission for an authorization name, enabling a user to create or access Big SQL resources. Privileges are stored in the catalogs. A role is an object that groups together one or more privileges and can be assigned to users, groups, PUBLIC or other roles with a GRANT statement. When roles are used, access permissions on Big SQL objects are associated with the roles. Users that are members of those roles gain the privileges defined for the role, which allows access to objects.

During SQL statement processing, the permissions that the Big SQL authorization model considers are a union of the following permissions:

1. The permissions granted to the primary authorization ID associated with the SQL statement
2. The permissions granted to the secondary authorization IDs (groups or roles) associated with the SQL statement
3. The permissions granted to PUBLIC, including roles that are granted to PUBLIC, directly or indirectly, through other roles

These features can be used (in conjunction with the Big SQL audit facility for monitoring access) to define and manage the level of security the installation requires.

Row and column access control (RCAC)

Row and column access control, or RCAC (sometimes referred to as fine-grained access control or FGAC), controls access to a table at the row level, column level or both. RCAC can be used to complement the table privileges model. To comply with various government regulations, users might implement procedures and methods to ensure that information is adequately protected. Individuals in the organization are permitted to access only the subset of data required to perform specific job tasks. For example, local government regulations might state that doctors are authorized to view the medical records of their own patients but not of other doctors' patients. The same regulations might also state that, unless a patient gives consent, a healthcare provider is not permitted to access the patient's personal information, such as a home phone number.

Row and column access control helps ensure that users have access to only the data that is required for their work. For example, a hospital system running Big SQL and RCAC can filter patient information and data to include only the data a particular doctor requires. Similarly, when patient service representatives query the patient table at the same hospital, they can view the patient name and telephone number columns but not the medical history column, which is masked for them. If data is masked, a NULL or alternate value is displayed instead of the actual medical history.

RCAC has the following advantages:

- No user is inherently exempted from the row and column access control rules.
- Table data is protected regardless of how a table is accessed via SQL. Applications, improvised query tools and report generation tools are all subject to RCAC. Enforcement is data-centric.
- No application changes are required to take advantage of this additional layer of data security. Row- and column-level access controls are defined in a way that is not apparent to existing applications. However, RCAC represents an important paradigm shift in that the key question becomes who is asking what instead of the traditional what is being asked. Result sets for the same query change according to the context in which the query is asked. There is no warning or error returned. This behavior is the point of the solution. It means that application designers and administrators must be conscious that queries do not see the whole picture (in terms of the data in the table) unless granted specific permission to do so.

Auditing

To manage access to sensitive data, users can deploy a variety of authentication and access control mechanisms that establish rules and controls for acceptable data access. To discover and protect against unknown or unacceptable behaviors, however, users can monitor data access with the Big SQL audit facility.

Successful monitoring of unwanted data access (and subsequent analysis) can help improve control of data access and, ultimately, help prevent malicious or careless unauthorized data access. The monitoring of application and individual user access can create a historical record of system activity. The Big SQL audit facility generates (and allows you to maintain) an audit trail for a series of predefined events. The records generated from this facility are kept in an audit log file. The analysis of these records can reveal usage patterns that would identify system misuse. Actions then can be taken to reduce or eliminate system misuse.

There are different categories of audit records that may be generated, including but not limited to the following:

- When audit settings are changed or when the audit log is accessed
- During authorization checks of attempts to access or manipulate Big SQL objects or functions
- When creating or dropping data objects and when altering certain objects
- When granting or revoking object privileges
- When altering the group authorization or role authorization
- When authenticating users or retrieving system security information
- During the execution of SQL statements

For any of the categories listed, users can audit failures, successes or both. Any operations on the Big SQL server may generate several records. The actual number of records generated in the audit log depends on the number of categories of events to be recorded as specified by the audit facility configuration. This number also depends on whether successes, failures or both are audited. For this reason, it is important to be selective about the events to audit.

Federation

Big SQL 3.0 supports federation to many data sources, including (but not limited to) DB2 for Linux, UNIX and Windows database software; IBM PureData™ System for Analytics; IBM PureData System for Operational Analytics; Teradata; and Oracle. This allows users to send distributed requests to multiple data sources within a single SQL statement.

To end users and client applications, data sources will appear as a single collective group in the Big SQL 3.0 server. Users and applications interface with the Big SQL server to access the data. The Big SQL server will contain a system catalog that stores information about the data that it can access. This catalog will contain entries that identify data sources and their characteristics. Big SQL uses routines stored in a library called a wrapper module to implement wrappers. Wrappers are mechanisms by which Big SQL interacts with data sources. One wrapper needs to be registered for each type of data source that will be accessed. Big SQL will consult the information stored in the catalog and the wrapper to determine the best plan for processing SQL statements. The Big SQL server processes SQL statements as if the data from all sources were ordinary tables or views within the Big SQL server.

It is important to note that characteristics of Big SQL take precedence when there are differences between these characteristics and the characteristics of the data source. Query results conform to Big SQL semantics, even if data from other sources is used to compute the query result. See the following examples:

- The code page that the Big SQL server uses is different from the code page the data source uses. In this case, character data from the source is converted according to the code page used by Big SQL when that data is returned to the user.
- The collating sequence that Big SQL uses is different from the collating sequence that the data source uses. In this case, any sort operations on character data are performed at the Big SQL server instead of at the data source.

Federation-aware query optimization

When a query is submitted, the Big SQL compiler consults information in the catalog and the data source wrapper to help it process the query. This includes details about connecting to the data source, server information, mappings and processing statistics. As part of the SQL compiler process, the query optimizer analyzes a query. The compiler develops alternative strategies, called access plans, for processing the query.

Access plans might call for the query to be one of the following:

- Processed by the data sources
- Processed by the Big SQL server
- Processed partly by the data sources and partly by the Big SQL server

The query optimizer evaluates the access plans primarily on the basis of information about the data source capabilities and the data. The wrapper and the catalog contain this information. The query optimizer decomposes the query into segments called query fragments. Typically, it is more efficient to push down a query fragment to a data source if the data source can process the fragment. However, the query optimizer takes into account other factors:

- Amount of data that needs to be processed
- Processing speed of the data source
- Amount of data the fragment will return
- Communication bandwidth

The query optimizer generates access plan alternatives for processing a query fragment. The plan alternatives perform varying amounts of work locally on the Big SQL server and on the remote data sources. Because the query optimizer is cost based, it assigns resource consumption costs to the access plan alternatives. The query optimizer then chooses the plan that will process the query for the lowest resource consumption cost.

If any of the fragments need to be processed by data sources, Big SQL submits these fragments to the data sources. After the data sources process the fragments, the results are retrieved and returned to the Big SQL server. If Big SQL performed any part of the processing, it combines its results with the results retrieved from the data source and then returns all results to the client.

Big SQL also provides the ability to process SQL that is not supported by a data source. This ability is called compensation. For example, Big SQL does not push down a query fragment if the data source cannot process it or if the Big SQL server can process it faster than the data source.

Big SQL compensates for lack of functionality at the data source in two ways:

- It can request that the data source use one or more operations equivalent to the Big SQL function stated in the query. For example, if a data source does not support the cotangent (COT(x)) function but does support the tangent (TAN(x)) function, Big SQL can request that the data source perform the calculation (1/TAN(x)), which is equivalent to the cotangent (COT(x)).
- It can return the set of data to the Big SQL server and perform the function locally.

With compensation, Big SQL can support the full SQL PL dialect for queries of data sources. Even data sources with weak SQL support or no SQL support will benefit from compensation.

Advanced monitoring

Monitoring performance in a distributed big data environment can be a challenge. Specifically, finding performance bottlenecks within a potentially huge number of data nodes may not be easy. Big SQL addresses this problem by providing a comprehensive set of metrics that help monitor the processing of queries as well as access to Hadoop. These metrics measure the volume read for Hadoop data and the time it took to be read. These metrics can be retrieved in real time by SQL table functions for each data node at different scopes, such as unit of work, table, SQL statement or globally. Because the metrics are based on SQL, it is possible to write monitoring queries or use SQL-based monitoring tools that aggregate the values of these metrics at different levels, giving users an overview of performance for the entire Hadoop cluster. Users are able to drill down to problem areas as well. In this way, it is possible to determine the underlying reasons for performance problems, such as slower read performance on certain data nodes or an uneven distribution of data across the data nodes.

5. Performance

Big SQL builds upon the vast experience of IBM in relational databases far beyond the richness of the available SQL. The expressiveness of query language and a wide array of available tools do not count for much if users can't get the information they need in a timely fashion. Big SQL is built on a foundation of technologies and experience from the extensive heritage IBM has in creating high-performance, highly scalable, robust and highly secure relational database platforms.

Query rewrites

Query optimization takes two forms: query rewrites and query plan optimization. Query plan optimization (discussed below) determines such things as the optimal order in which tables will be accessed and the most efficient join strategy for implementing the query. However, before plan optimization can take place, query rewrite optimization must be performed.

Query rewriting is exactly what the name implies: automatically determining the optimal way in which a query could be expressed in order to dramatically reduce the resources necessary to execute it. For example, if the same expression appears multiple times in the same query, the query may be rewritten to compute the result of the expression once and reuse the value in multiple locations or even to take multiple references to the same table and determine if the query can be expressed in a way that allows the table to be scanned only once to satisfy all of the references.

The Big SQL query rewrite engine consists of more than 140 distinct rewrite rules. The following subsections highlight a few such optimizations.

Subquery optimization

Subqueries are the bane of join processing in Hadoop. For a system in which data is randomly scattered across the cluster and indexes are generally unavailable, nested loop joins (in which each row in the outer table results in a scan of the inner table) can be tremendously expensive. If left in place, subqueries will frequently result in nested loop joins. This is why a major aspect of query rewrites is finding another way to express a subquery.

Such optimizations include:

Subquery to join

Certain subqueries may be reexpressed as a single join operation. For example:

```
SELECT PS.*
FROM TPCD.PARTSUPP PS
WHERE PS.PS_PARTKEY IN
  (SELECT P_PARTKEY
  FROM TPCD.PARTS
  WHERE P_NAME LIKE 'FOREST%')
```

May instead be expressed as:

```
SELECT PS.*
FROM PARTS, PARTSUPP PS
WHERE PS.PS_PARTKEY = P_PARTKEY AND
  P_NAME LIKE `FOREST%'
```

Subquery decorrelation

Wherever possible, correlated subqueries are expressed as derived tables. As a derived table, the subquery may be joined using a hash or redistribution join. For example, the following query:

```
SELECT SNO AS C1, PNO AS C2, JNO AS C3, QTY AS C4
FROM SUPPLY SP1
WHERE QTY = (
  SELECT MAX(QTY)
  FROM SUPPLY SP2
  WHERE SP1.SNO = SP2.SNO
  AND SP2.PNO <> 'P2')
```

Can instead be written as:

```
SELECT SNO AS C1, PNO AS C2, JNO AS C3, QTY AS C4
FROM SUPPLY SP1,
  (SELECT MAX(QTY) C0, SP2.SNO C1
  FROM SUPPLY SP2
  WHERE SP2.PNO <> 'P2'
  GROUP BY SP2.SNO) T1
WHERE QTY = T1.C0
AND T1.C1 = SP1.SNO
```

Shared aggregation

For queries involving multiple aggregations, resources can be saved by combining aggregations together, such as taking the following query:

```
SELECT SUM(O_TOTAL_PRICE) AS OSUM,
  AVG(O_TOTAL_PRICE) AS OAVG
FROM ORDERS
```

And rewriting it as:

```
SELECT OSUM, OSUM/OCOUNT AS OAVG
FROM (SELECT SUM(O_TOTAL_PRICE) AS OSUM,
  COUNT(O_TOTAL_PRICE) AS OCOUNT
  FROM ORDERS) AS SHARED_AGG
```

Predicate pushdown

Wherever possible, Big SQL attempts to push static argument predicates as close to where the data is processed as possible. (In fact, as described in the Architecture section, predicates can be pushed all the way down into the physical read of the data from disk.) For example, given a view such as:

```
CREATE VIEW LINEITEM_GROUP(SUPPKEY, PARTKEY, TOTAL)
AS SELECT L_SUPPKEY, L_PARTKEY, SUM(QUANTITY)
  FROM TPCD.LINEITEM
  GROUP BY L_SUPPKEY, L_PARTKEY
```

And a query such as:

```
SELECT *
FROM LINEITEM_GROUP
WHERE SUPPKEY = 1234567
```

Big SQL will combine the view definition and the query together and push the search predicate inside of the view's group, eliminating rows before they are ever grouped together:

```
SELECT L_SUPPKEY, L_PARTKEY, SUM(QUANTITY)
FROM TPCD.LINEITEM
WHERE L_SUPPKEY = 1234567
GROUP BY L_SUPPKEY, L_PARTKEY
```

Distinct elimination

A number of query rewrites leverages the presence of referential integrity constraints, such as primary or unique key definitions or CHECK constraints. For example, given a query such as:

```
SELECT DISTINCT CUSTKEY, NAME FROM CUSTOMER
```

If the CUSTOMER table has a primary or unique key on the custkey column, Big SQL will automatically eliminate the need for the DISTINCT operation:

```
SELECT CUSTKEY, NAME FROM CUSTOMER
```

Statistics-driven optimization

Big SQL maintains a rich set of table-level, partition-level and column-level statistics that are utilized by its query optimizer to determine the best approach to take for executing a query. These include:

- Operation order
  - Join order
  - Order of predicate application
  - Aggregation strategy (e.g., partial aggregation at each node vs. final aggregation)

- Join implementation
  - Nested loop join
  - Sort-merge join
  - Hash join

- Location
  - Repartition or broadcast join
  - Multi-core parallelism (apply more resources for one query vs. multiple queries)
  - Federated pushdown operations or compensation

When evaluating the query execution plan, Big SQL considers much more than the object-level statistics available for each table in the query. And it takes into account the number of rows produced by each operator involved in the query, the CPU, I/O and memory costs for each operator and the communication costs between nodes (both for the Hadoop cluster and federated environments). It is designed to produce a query plan that minimizes total resource consumption while optimizing for minimum elapsed time.

Conclusion

It is understandable why there has been such an explosion of SQL-on-Hadoop technologies over the past few years. Hadoop offers an inexpensive, highly scalable solution for tackling the ever-increasing volumes and varieties of data that organizations must contend with in the era of big data. While Hadoop provides the framework with which to scale the data, SQL offers the ability to expose that data, using languages that employees already understand, as well as the ability to leverage the enormous ecosystem of tools built around the SQL standards and the relational model that IBM spearheaded in the 1970s and continues to expand today.

Big SQL 3.0 is the first solution to tackle the full breadth and depth of the SQL language, along with enterprise-class features, performance and security.

For more information

Learn more about IBM InfoSphere BigInsights for Hadoop and Big SQL 3.0:

ibm.com/software/data/infosphere/biginsights

SWW14019-USEN-00