S

Spark SQL

Xiao Li, Cheng Lian, and Shu Mo
Databricks Inc, San Francisco, CA, USA

Definition

SQL is a highly scalable and efficient relational processing engine with ease-to-use APIs and mid-query fault tolerance. It is a core module of Apache Spark, which is a unified engine for distributed data processing (Zaharia et al. 2012). Spark SQL can process, integrate, and analyze the data from diverse data sources (e.g., Hive, Cassandra, Kafka, and Oracle) and file formats (e.g., Parquet, ORC, CSV, and JSON). The common use cases include ad hoc analysis, logical warehouse, query federation, and ETL processing. It also powers the other Spark libraries, including structured streaming for stream processing, MLlib for machine learning (Meng et al. 2016; Michael et al. 2018), GraphFrame for graph-parallel computation (Dave et al. 2016), and TensorFrames for TensorFlow binding. These libraries and Spark SQL can be seamlessly combined in the same application with holistic optimization by Spark SQL.

Overview

Spark is a general purpose big data processing system. It was originally developed in the AMPLab at UC Berkeley and donated to Apache Software Foundation in 2013. Now, Apache Spark is one of the most popular open-source projects in data analytics and query processing. Spark SQL (Armbrust et al. 2015) (its predecessor, Shark (Xin et al. 2013)) was introduced in 2014 and became the core of Apache Spark ecosystem. It enables Spark to perform efficient and fault-tolerant relational query processing with analytics database technologies. The relational queries are compiled to Resilient Distributed Dataset (RDD) transformations and actions, which are executable in Spark.

Spark SQL follows the classic query processing and federation architecture with adoption of the recent research (e.g., whole-stage code generation). Through the user-facing APIs (SQL, DataFrame, and Dataset), the user queries are converted to unresolved abstract syntax trees (called unresolved logical plans). The plans are then analyzed using the session-specific temporary view manager, and the cross-session cache manager and catalog. Logical optimizer and physical planner optimize the resolved plans by applying heuristics-based and cost-based transformation rules. During the query planning phase, the sub-plans are pushed down to the underlying data sources for more efficient processing, if possible; the logical plans are converted to the executable physical plans consisting of transformations and actions on RDDs with the generated Java code. The code is compiled to Java bytecode, executed at runtime

S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_251-1

by JVM and optimized by JIT to native machine code at runtime.

Declarative APIs

Prior to Spark SQL, Spark offers low-level procedural APIs for operating RDDs and building DAGs that are executable in Spark. Spark SQL introduces three declarative APIs (DataFrame, Dataset, SQL language), which are complementary to the low-level Spark APIs (i.e., RDD APIs). It facilitates tight integration of relational queries and complex procedural processing.

The SQL API is based on ANSI SQL:2003 standard with full compliance to Hive query language (HiveQL). For the existing Hive users, the compliance facilitates migration to Spark SQL. Spark SQL also provides language-integrated and lazily evaluated DataFrame/-Dataset APIs. The DataFrame API provides untyped relational operations, while the Dataset API provides a typed version for better type safety. The DataFrame API is available in Scala, Java, Python, and R. The Dataset API is only available in Scala and Java since Python and R are dynamically typed languages.

Compared with SQL, DataFrame/Dataset APIs provide richer and user-friendly interfaces, since the APIs are not limited by ANSI SQL compliance and also fully integrated with the programming languages (Java/Scala/Python/R). Using DataFrame/Dataset APIs, a complex dataflow logics can be split to multiple simpler modular host-language functions and then build up a single logical plan for holistic query optimization. All these three APIs are internally represented by the same Catalyst logical plans. They can be mixed, combined, optimized, and executed holistically, thanks to the lazy execution. The execution is triggered until users call the action APIs (e.g., collect, save, and show).

Query Optimization

Spark SQL optimizes the plans using the stratified search strategy that are widely used in the commercial database vendors (e.g., Oracle and DB2). First, the optimizer rewrites the query plans using heuristics-based rules. The typical transformation rules include predicate pushdown, column pruning, outer join elimination, constraint propagation, and predicate inference. Then, the cost-based plan enumeration and method selection are executed. The cost-based optimizer framework is initially shipped with Spark 2.2 and still rapidly evolving. Histogram was introduced for cardinality estimation in Spark 2.3. More accurate cost model, demand-driven enumerators, and adaptive query optimization are in the road map.

Spark SQL optimizer is extensible. Custom optimizer rules can be injected. For example, the users can add extra optimization rules for pushing more operators into the external data source systems or supporting the new data types.

Query Execution

Memory and CPU, rather than disk and network I/O, are the major performance bottlenecks of query execution in Apache Spark (Ousterhout et al. 2017), thanks to the progress in related hardware and data compression. The project Tungsten speeds up query execution by optimizing the efficiency of CPU and memory. The major focuses include off-heap memory management and runtime code generation.

Memory Management

The overhead of JVM objects and GC are significant for data intensive Java applications. Apache Spark leverages the application semantics to explicitly manage the memory using the *sun.misc.Unsafe* feature. This feature provides the C-style direct access to off-heap memory. The memory managed by Spark is invisible to the garbage collector. Tuning GC for higher performance is not needed. The compiled/interpreted operations directly manipulate the binary data represented in the Tungsten specialized format. Compared with JVM objects, it has less memory footprint and thus reduces the processing time.

Runtime Code Generation

On the driver, Spark generates Java source code specialized to each query. On the executors, the generated code is compiled to Java bytecode by a lightweight Java compiler, which runs directly on the JVM and can be further compiled to native code by the just-in-time (JIT) compiler in the JVM. It strikes a good balance between performance and readability (and thus debuggability) of generated code.

Runtime code generation is performed on two levels: (1) expressions are generated into straightforward Java code to reduce interpretation overhead, and (2) where possible, multiple adjacent physical plan operators, along with the expressions involved, are fused together using a push-based model and generated into a single code generation unit (called a stage). Compared with the original iterator-based pull model (Volcano), this reduces the overhead of virtual function calls and materialization of intermediate results between operators, improves data locality, and enables further specialization with the context of multiple operators.

The same code generation framework is also used to speed up serialization/deserialization. As a unified data processing framework, Spark SQL supports flexible use of UDFs and type-safe Dataset APIs, both of which involve conversions between domain objects and Spark SQL internal data representations.

Data Sources

Spark separates computation and storage. The data sources are autonomous and can be shared with the other processing engines. The data schema is dynamic. The schema is just a virtual view that can be predefined or derived when reading it. For example, the built-in file source connectors, including JSON, CSV, Parquet, and ORC, offer read-time schema inference. All the inferred or predefined schemas can be stored in a global persistent catalog, which is Hive metastore by default. The Hive metastore can also be shared with the other engines (e.g., Hive).

Thanks to the rich interoperability with external data sources, Spark SQL can read from, integrate with, and write to a variety of data sources. Users can use the built-in connectors and also plugin other third-party connectors. Through the data source APIs, third parties can build customized connectors to access the data stores.

Highly efficient vectorized readers are provided for columnar file sources (e.g., Parquet and ORC). Such complicated I/O operations are not fused into the whole-stage codegen. Bulk reading and processing can reduce the per-tuple interpretation overhead and leverage compiler optimization. The built-in cache mechanism is also columnar. The external sources and intermediate results can be explicitly cached in memory or local disk for reuse.

Conclusion

Since the initial release, Apache Spark has quickly become the largest open-source community in big data. It is the work of over 1000 contributors from over 250 companies. Spark SQL is the most active component in Apache Spark. Spark SQL brings end-to-end optimization in the sophisticated applications, which use one or multiple Spark libraries (e.g., SQL, MLlib, and structured streaming). More breakthroughs are expected in the coming years, as Spark SQL is growing to be a compiler of the unified analytics engine.

Cross-References

▸ In-Memory Big Data Processing

References

Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M (2015) Spark SQL: relational data processing in spark. In: Proceedings of the ACM SIGMOD international conference on management of data (SIGMOD'15)

Dave A, Jindal A, Li LE, Xin R, Gonzalez J, Zaharia M (2016) Graphframes: an integrated API for mixing

graph and relational queries. In: Proceedings of the 4th international workshop on graph data management experiences and systems (GRADES'16)

Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A (2016) MLlib: machine learning in apache spark. J Mach Learn Res 17(1):34:1–34:7

Michael A, Tathagata D, Joseph T, Burak Y, Shixiong Z, Reynold X, Ali G, Ion S, and Matei Z (2018) Structured Streaming: A declarative API for real-rime applications in apache spark. In: Proceedings of the ACM SIGMOD international conference on management of data (SIGMOD'18). 601–613

Ousterhout K, Canel C, Ratnasamy S, Shenker S (2017) Monotasks: architecting for performance clarity in data analytics frameworks. In: Proceedings of the 26th ACM symposium on operating system principles

Xin RS, Rosen J, Zaharia M, Franklin MJ, Shenker S, Stoica I (2013) Shark: SQL and rich analytics at scale. In: Proceedings of the ACM SIGMOD workshop on the web and databases (SIGMOD'13)

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX symposium on networked systems design & implementation (NSDI'12)