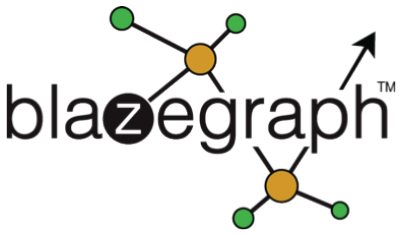


Introduction to Blazegraph Database

An ultra-scalable, high-performance graph database

About this White Paper Series

Blazegraph, which was founded in 2006 as SYSTAP, created the industry's first GPU-accelerated high-performance database for large graphs. The company's software is designed for solving complex graph and machine learning algorithms. This three-part white paper series will provide a comprehensive overview of the company's core product – Blazegraph Database, which is the platform for a family of Blazegraph products for graph applications at large scale ranging from an Enterprise Edition with High Availability (HA) and scale-out to GPU Acceleration.

This first white paper will provide an introduction to the product. Other papers in this series will focus on the Blazegraph Database's scale-up and scale-out architectures and unique features.

Table of Contents

Introduction

The Blazegraph Database is an ultra-scalable, high-performance graph database with support for the Blueprints and Semantic Web APIs, capable of supporting up to 50 billion edges on a single machine. Written entirely in Java, the platform supports both the property graph and Resource Description Framework (RDF)[1] data models, as well as the SPARQL 1.1[2] family of specifications, including query update, basic federated query, and service description.

Blazegraph Database also offers support for novel extensions for durable named solution sets, efficient storage and querying of reified statement models, scalable graph analytics and GPU-accelerated graph processing. The database enables multi-tenancy and can be deployed as an embedded database, a standalone server, a highly available replication cluster, or a horizontally sharded federation of services similar to Google's Bigtable or Accumulo and Cassandra from Apache.

The Community Edition Blazegraph open source platform has been under continuous development since 2006, is available under a dual licensing model (GNU General Purpose License, Version 2 [GPLv2][3] and commercial licensing). Enterprise features for high availability and scale-out, as well as GPU acceleration, are available under a commercial license. A number of well-known companies also OEM, resell or embed the Blazegraph Database in their applications.

Blazegraph[4] leads the development of product and offers support subscriptions for both commercial and open-source users. Our goal is a robust, scalable, high-performance, and innovative platform.

Blazegraph Database Architecture

Blazegraph Database[5, 6] is a horizontally scaled, general-purpose storage and computing fabric for ordered data (B+Trees), designed to operate on a cluster of commodity hardware. While many clustered databases rely on a fixed, and often capacity-limited, hash-partitioning architecture, Blazegraph Database uses dynamically partitioned key-range shards. This architecture was chosen to remove any realistic scaling limits. In principle, Blazegraph Database may be deployed on tens, hundreds, or even thousands of machines. Unlike hash-partitioned approaches, though, with Blazegraph Database, new capacity may be added incrementally to data centers without requiring the full reload of all data.

[1] https://www.w3.org/RDF/
[2] https://www.w3.org/TR/sparql11-overview/
[3] http://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html
[4] Formerly known as SYSTAP, LLC, https://www.blazegraph.com/press/#systapchanges
[5] http://www.blazegraph.com/
[6] https://github.com/blazegraph/database

On top of that core is the Blazegraph RDF Store, a massively scalable RDF database supporting RDFS and OWL Lite reasoning, high-level query (SPARQL), and datum level provenance.

Deployment Models

Blazegraph Database supports several distinct deployment models with both scale-up and scale-out architectures:

- Embedded (Community Edition)
- Single Server (Community Edition)
- High Availability (Enterprise Edition)
- Federation [Scale-out] (Enterprise Edition)
- GPU Acceleration[7] (Commercial)

These deployment models are based on two distinct architectures: scale-up and scale-out[8]. The embedded database, WAR (Single Server), and the replication cluster are *scale-up* architectures based on the journal. The Blazegraph Community edition supports the embedded and single Th de t for index management against a single backing store. The Enterprise edition also supports the replication cluster scale-up model (High Availability). The Blazegraph Enterprise edition also supports federation, which is a *scale-out* architecture using dynamically partitioned indices to distribute the data within each index across the resources of a compute cluster. All deployment models support the Storage And Inference Layer (SAIL), SPARQL 1.1 Query and SPARQL Update.

The benefits of the scale-out architecture are significant. Using the scale-out architecture, a cluster can scale to petabytes of data and has much greater throughput than a single machine. However, scale-out has higher latency for selective queries due to the increased overhead of internode communication. Updates on the journal and replication cluster are ACID[9], but updates on the federation are shard-wise ACID. Finally, while it is always important to vector operations against indices, vectored operations are absolutely required for good performance on the scale-out architecture.

[7] https://www.blazegraph.com/product/gpu-accelerated/

[8] Scale-up and scale-out architecture will be discussed more in depth in the second white paper in this series.

[9] "ACID is an acronym for four common database properties: Atomicity, Consistency, Isolation, Durability." Reuter, Andreas; Haerder, Theo (December 1983). "Principles of Transaction-Oriented Database Recovery." ACM Computing Surveys (ACSUR) 15 (4): 287-317.

The choice of the right deployment model depends on your requirements. The journal offers low latency operations because of its *locality*. It scales to ~50B triples or quads on a single machine and offers a low total cost of ownership. The replication cluster retains the architecture of the journal, but adds high availability and horizontal scaling of query (but not data) without sacrificing the write performance of the database. The federation has higher latency, resulting from the overhead of internode coordination, and offers greater throughput for some query workloads. The federation can scale-out far beyond the journal. But, because of higher coordination costs, at least three machines are needed to deliver performance similar to a single-machine journal. By having the low-latency of the journal combined with high availability and horizontally scaled query on a three-node replication cluster, the scale-out architecture of the federation really makes sense for very large data sets and clusters of eight or more machines.

Concurrency Control

Blazegraph supports optional transactions based on Multiversion Concurrency Control (MVCC)[10]. Many database architectures are based on two-phase locking (2PL), which is a pessimistic concurrency control strategy. In 2PL, a transaction acquires locks as it executes and readers and writers will block in their access conflicts with the locks for running transactions. MVCC is an optimistic concurrency control strategy and relies on the use of timestamps to detect conflicts when a transaction is validated. MVCC enables very high concurrency since readers never block and writers can run concurrently even when they touch the same region of the disk (there is no sense of a row, page or table lock). If two writers modify the same tuple in an index, then that conflict is detected when the transaction validates and the second transaction will fail unless the conflict can be resolved. The Blazegraph Database can resolve many of these write-write conflicts for RDF. The MVCC design and the ability to choose whether or not operations will be isolatable by transactions is driven deep into the architecture, including the copy-on-write mechanisms of the B+Tree, the journal and backing store architectures, and the history retention policy.

Transaction processing on a federation is optional by design. Transactions can greatly simplify application architecture, but they can limit both performance and scale through increased coordination costs. For example, Google developed its “row store”[11] to address a set of very specific application requirements. In particular, Google had a requirement for extremely high concurrent read writes and very high concurrent write rates. Distributed transaction processing was ruled out because each commit must be coordinated with the transaction service, which limits the potential throughput of a distributed database. In its design, Google opted to restrict concurrency control to ACID operations on “rows” within a

[10] “Naming and Synchronization in a Decentralized Computer System.” Reed, D.P. MIT dissertation. http://www.lcs.mit.edu/publications/specpub.php?id=773
[11] “Bigtable: A Distributed Storage System for Structured Data,” http://research.google.com/archive/bigtable-osdi06.pdf

"column family." With this design, a purely local locking scheme may be used, which will enable a substantially higher concurrency. Blazegraph DB uses this approach for its "row store," for the lexicon for an RDF database and for high throughput distributed bulk data load.

> *"Blazegraph provides an immortal database architecture with a configurable history retention policy."*

For a federation, distributed transactions[12] are primarily used to support snapshot isolation for query. An "isolatable" index (one that supports transactional isolation) maintains per-tuple revision timestamps, which are used to detect and, when possible, reconcile write-write conflicts. The transaction service is responsible for assigning transaction identifiers, which are timestamps, revision timestamps and commit timestamps. The transaction service maintains a record of the open transactions and manages read-locks on the historical states of the database. The read-lock is just the timestamp of the earliest running transaction, but it plays an important role in managing resources, which will be discussed later in this paper.

Managing Database History

The Blazegraph Database provides an *immortal database* architecture with a configurable *history retention policy*. An immortal database is one in which you can request a consistent view of the database at any point in its history, which essentially enables you to wind back the clock to the state of the database at some prior day, month or year. This feature can be used in many interesting ways, including for regulatory compliance and examining changes in the state of accounts over time.

For many applications, access to unlimited history is not required. Therefore you can configure the amount of history that will be retained by the database. To configure, you specify the minimum age before a commit point may be released. This age can be five minutes, one day, two weeks or 12 months. The minimum release age also can be set to zero, in which case the Blazegraph Database will release the resources associated with historical commit points as soon as the read locks for those resources have been released. Equally, the minimum age can be set to a very large number, in which case historical commit points will never be released.

The minimum release age determines which historical states you can access, not the age of the oldest record in the database. For example, if you have a five-day history retention policy, and you insert an entry, or tuple, into an index, then that tuple would remain in the index until five days after it was

[12] Blazegraph supports both read-only and read-write transactions in its single server mode and HA replication cluster, and distributed read-only transactions on a federation. Distributed read-only transactions are used for query and when computing the closure over an RDF database. Support for distributed read-write transactions on a federation has been contemplated, but never implemented.

overwritten or *deleted*. If you never update that tuple, the original value will never be released. If you do delete the tuple, then you will still be able to read from historical database states containing that tuple for the next five days. Applications can apply additional logic if they want to delete records once they reach a certain age; this can be done efficiently in terms of the tuple revision timestamps.

B+Trees

The B+Tree is a central data structure for database systems because it provides search, insert and update in logarithmic amortized time. The Blazegraph Database B+Tree fully implements the tree balancing operations and remains balanced under inserts and deletes. The mutable B+Tree implementation is single threaded under mutation, but allows concurrent readers. In general, readers do not use the mutable view of a B+Tree, so readers do not block for writers. Figure 1 shows the Blazegraph B+Tree architecture.

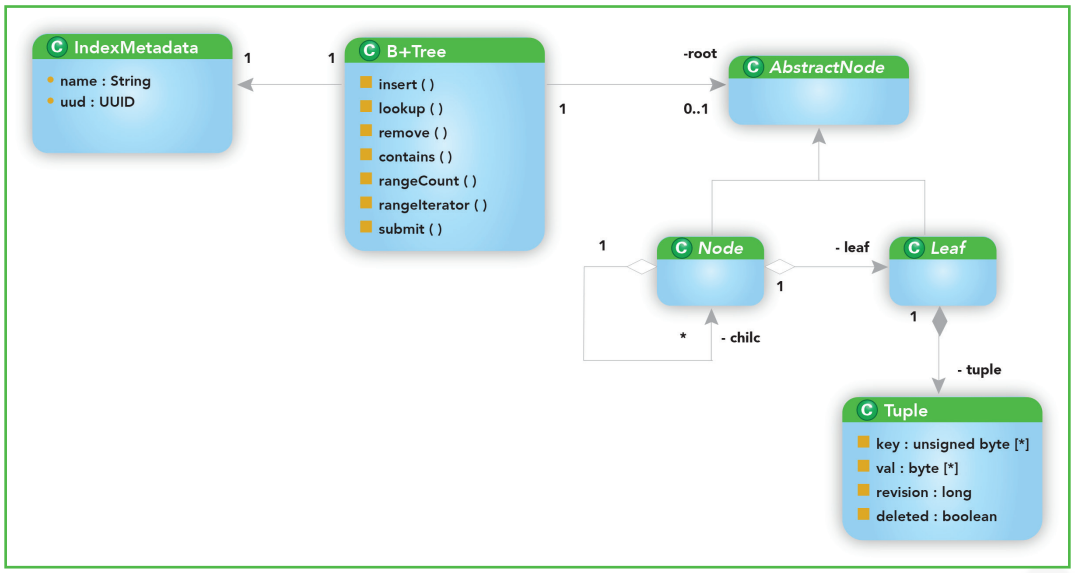


Figure 1 – B+Tree architecture.

For scale-out, each B+Tree key-range partition is a view comprised of a mutable B+Tree instance with zero or more read-optimized, read-only B+Tree files known as index segments. The index segment files support fast double-linked navigation between leaves – they are used to support the dynamic sharding process on a federation. It uses a constant (and configurable) branching factor and enables the page size of the index to vary, which works out well with overall copy-on-write architecture and simplifies some decisions in the maintenance of the index.

In Blazegraph Database, an index maps unsigned byte[ ] keys to byte[ ] values. Mechanisms are provided to support the encoding of single and multi-field numeric, ASCII and Unicode data. Likewise,

extensible mechanisms provide for (de)serialization of application data as byte[ ]s for values. An index entry is known as a tuple. In addition to the key and value, a tuple contains a "deleted" flag that is used to prevent reads through to historical data in index views, discussed below, and a revision timestamp, which supports optional transaction processing based on MVCC. The IndexMetadata object is used to configure both local and scale-out indices. Some of its most important attributes are the index name, index UUID, branching factor, objects that know how to serialize application keys and both serialize and deserialize application values store in the index, and the key and value coder objects.

The B+Tree never overwrites records (nodes or leaves) on the disk. Instead, it uses copy-on-write for clean records, expands them into Java objects for fast mutation and places them onto a hard reference ring buffer for that B+Tree instance. On eviction from the ring buffer, and during checkpoint operations, records are coded into their binary format and written on the backing store.

Records can be directly accessed in their coded form. The default key coding technique is front coding, which supports fast binary search with good compression. Canonical Huffman[13, 14] coding is supported for values. Custom coders may be defined, and can be significantly faster for specific applications.

The high-level API for the B+Tree includes methods that operate on a single key-value pair (insert, lookup, contains, remove) or on key ranges (rangeCount, RangeIterator), and a set of methods to submit Java procedures that are mapped against the index and executed locally on the appropriate data services for the scale-out architecture. Scale-out applications make extensive use of the key-range methods, mapped index procedures and asynchronous write buffers to ensure high performance with distributed data.

The *rangeCount(fromKey,toKey)* method is of particular relevance for query planning. The B+Tree nodes internally track the number of tuples spanned by a separator key. Using this information, the B+Tree can report the cardinality of a key-range on an index using only two key probes against the index. This range count will be exact, unless delete markers are being used, in which case it will be an upper bound (the range count includes the tuples with delete markers). Fast range counts also are available on a federation, where a key-range may span multiple index partitions.

[13] Huffman coding, http://en.wikipedia.org/wiki/Huffman_coding
[14] Canonical Huffman coding, http://en.wikipedia.org/wiki/Canonical_Huffman_code

RDF Database Architecture

In this section, the RDF is defined and we show how an RDF database is realized using the Blazegraph Database architecture. The Blazegraph Database implements the SAIL API, which provides a pluggable backend for the Sesame (RDF4J) platform[15]. However, the query evaluation and transaction models for the Blazegraph Database differ significantly from those of the Sesame platform.

RDF Defined

The RDF[16, 17] may be understood as a general-purpose, schema-flexible model for describing metadata and graph-shaped information. RDF represents information in the form of statements (triples or quads). Each triple connotes an edge between two nodes in a graph. The quad position can be used to give statements identity (The Blazegraph Database provenance mechanism is based on this approach) or to place statements within a named graph.

RDF provides some basic concepts used to model information – statements are composed of a subject (a Uniform Resource Identifier (URI) or a Blank Node), a predicate (always a URI), an object (a URI, Blank Node, or Literal value), and a context (a URI or a Blank Node). URIs are used to identity a particular resource[18], whereas Literal values describe constants, such as character strings, and may carry either a language code or data type attribute in addition to their value. RDF also provides an XML-based syntax (called RDF/XML[19]) for interchanging RDF graphs.

There is also a model theoretic layer above the RDF model and RDF/XML interchange syntax that is useful for describing ontologies and for inference. RDF Schema[20] and the OWL Ontology Web Language[21] (OWL) are two such standards-based layers on top of RDF. RDF Schema is useful for describing class and property hierarchies. OWL is a more expressive model. Specific OWL constructs may be applied to federation and semantic alignment, such as owl:equivalentClass and owl:equivalentProperty (for aligning schemas) and owl:sameAs (for dynamically snapping instance data together).

There is an inherent tension between expressivity and scale, since high expressivity is computationally expensive and gets more so as data size increases. Blazegraph Database has focused on scale over expressivity.

[15] Sesame, http://rdf4j.org/
[16] Resource Description Framework, http://www.w3.org/RDF/
[17] RDF Semantics, http://www.w3.org/TR/rdf-mt/
[18] The benefit of URIs over traditional identifiers is twofold. First, by using URIs, RDF may be used to describe addressable information resources on the Web. Second, URIs may be assigned within namespaces corresponding to Internet domain, which provides a decentralized mechanism for coining identifiers.
[19] http://www.w3.org/TR/rdf-syntax-grammar/
[20] http://www.w3.org/TR/rdf-schema/
[21] http://www.w3.org/2004/OWL/
[22] Blazegraph uses Reification Done Right (RDR) support to implement provenance: http://arxiv.org/pdf/1406.3399.pdf

Database Schema for the RDF

Blazegraph supports three distinct RDF database modes: triples, triples with provenance[22] and quads. These modes reflect slight variations on a common database schema. Abstractly, this schema can be conceptualized as a **lexicon** and a **statement** relation, each of which uses several indices. The ensemble of these indices is collectively an RDF database instance. Each RDF database is identified by its own namespace. Any number of RDF database instances may be managed within a Blazegraph instance.

Lexicon

A wide variety of approaches have been used to manage the variable length attribute values, arbitrary cardinality of attribute values and the lack of static typing associated with RDF data. Blazegraph uses a combination of inline representations for numeric and fixed length RDF Literals with dictionary encoding of URIs and other Literals. The inline representation is typically one byte larger than the corresponding primitive data type and imposes the natural sort order for the corresponding data type. Inline representations for xsd:decimal and xsd:integer use a variable length encoding. URIs declared in a vocabulary when the Knowledge Base (KB) instance was created are also inlined (in 2-3 bytes). Depending on the configuration, blank nodes are typically inlined. Statements about statements are inlined as the representation of the statement they describe.

The encoded forms of the RDF Values are known as *Internal Values* (IVs). IVs are variable length identifiers that capture various distinctions that are relevant to both RDF data and how the database encodes RDF values. Each IV includes a flags byte that indicates the kind of RDF value (URI, Literal, or Blank node), the natural data type of the RDF value (Unicode, xsd:byte, xsd:short, xsd:int, xsd:long, xsd:float, xsd:double, xsd:integer, etc.), whether the RDF Value is entirely captured by an *inline* representation, and whether this is an *extension* data type. User defined data types can be created using an *extension byte* that optionally follows the flags byte. Inlining is used to reduce the stride in the statement indices and to minimize the need to materialize RDF values out of the dictionary indices when evaluating SPARQL FILTERs.

The lexicon is comprised of three indices:

- Binary Large Objects (BLOBS) – Large Literals and URIs are stored in a BLOBS index. The key is formed from a flags byte, an extension byte, the int32 hash code of the Literal, and an int16 collision counter. The value associated with each key is the Unicode representation of the RDF value. The use of this index helps to keep very large literals out of the TERM2ID index where they can introduce severe skew into the B+Tree page size. The hash code component of the

[22] Blazegraph uses Reification Done Right (RDR) support to implement provenance: http://arxiv.org/pdf/1406.3399.pdf

BLOBS index introduces significant random IO during load operations. Therefore, the use of the BLOBS index is limited to Literals whose string length is over a threshold (256). This is only a small percentage of the Literals in the data sets that we have examined.

- TERM2ID – The key is the Unicode collation key for the Literal or URI. The value is the assigned int64 unique identifier.
- ID2TERM – The key is the identifier (from the TERM2ID index). The value is the RDF value.

Writes on the lexicon indices use an eventually consistent approach, which enables lexicon writes to be made without global locking in a federation. An optional full text index maps tokens extracted from RDF values onto the internal identifiers for those RDF values and may be used to perform a keyword search against the triple or quad store.

Statement Indices

The statement relation models the subject, predicate, object and, optionally, the context for each statement. The RDF database uses covering indices as first described in YARS[23]. For each possible combination of variables and constants in a basic triple pattern (or quad pattern), there is a clustered index that has good locality for that access pattern. A triple store requires three statement indices (SPO, POS and OSP). A quad store requires six statement indices (OCSP, SPOC, CSPO, PCSO, POCS and SPOC). In each case, the name of the index indicates the manner in which the Subject, Predicate, Object, and the optional Context have been ordered to form the keys for the index.

SPARQL Query Processing

It is important to keep in mind the architectural differences between the scale-up architecture (including the Journal, the WAR and the HA replication cluster) and the scale-out architecture (the federation). Index scans on the scale-up architecture turn into random IOs since the index is not in key order on the disk. However, index scans on the scale-out architecture turn into sequential IOs, as the vast majority of all data in a cluster is on read-only index segment files in key order on the disk. This architectural difference means that a cluster is able to more efficiently handle query plans that do sustained index scans. However, since index scans turn into random IO on the scale-up architecture, you should use either lots of spindles or Solid State Drives (SSD) to reduce the IO Wait for the disk.

In addition to the inherent resources and opportunities for increased parallelism, the federation has two other architectural benefits. First, the scale-out architecture can use a bloom filter in front of each

[23] Andreas Harth, Stefan Decker. "Optimized Index Structures for Querying RDF from the Web." 3rd Latin American Web Congress, Buenos Aires - Argentina, Oct. 31 - Nov. 2 2005.

index segment. This means that point tests can be much faster on a cluster than on a single machine since correct rejections will never touch the disk. Second, all B+Tree nodes in an index segment are in one contiguous region on the disk. When the index segment is opened, the nodes are read in using a single sustained IO. Thereafter, a read to a leaf on an index segment will perform at most one IO.

RDF query is based on statement patterns. A triple pattern has the general form (S,P,O), where S, P and O are either variables or constants in the subject, predicate, and object position respectively. For the quad store, this is generalized as patterns having the form (S,P,O,C), where C is the context (or graph) position and may be either a blank node or a URI. Blazegraph translates SPARQL into an Abstract Syntax Tree (AST) that is fairly close to the SPARQL syntax and then applies a series of rewrite optimizers on that AST.

Those optimizers handle a wide range of problems, including:

- substituting constants into the query plan;
- generating the WHERE clause and projection for a DESCRIBE or CONSTRUCT query;
- static analysis of variables;
- flattening of groups;
- elimination of expressions or groups that are known to evaluate to a constant;
- ensuring that query plans are consistent with the bottom-up evaluation semantics of SPARQL;
- reordering joins; and
- attaching FILTERS in the most advantageous locations.

The rewrites are based on either fully decidable criteria or heuristics, rather than searching the space of possible plans. The use of heuristics makes it possible to answer queries having 50 to 100 joins with very low latency – as long as the joins make the query selective in the data. Joins are re-ordered based on a static analysis of the query, the propagation of variable bindings, fast cardinality estimates for the triple patterns, and an analysis of the propagation of in-scope variables between sub-groups and sub-SELECTs.

Once the AST has been rewritten, it is translated into a physical query plan. Each group graph pattern surviving from the original SPARQL query will be modeled by a sequence of physical operators. Nested groups are evaluated using solution set hash joins. Visibility of variables within groups and sub-queries adhere to the rules for variable scope for SPARQL (e.g., as if bottom up evaluation were being

performed). For a given group, there is generally a sequence of required joins corresponding to the statement patterns in the original query. There also may be optional joins, sub-SELECT joins, and joins of pre-computed named solution sets.

Constraints (FILTERs) are evaluated as soon as the variables involved in the constraint are known to be bound and no later than the end of the group. Many SPARQL FILTERs can operate directly on IVs. When a FILTER requires access to the materialized RDF Value, the query plan includes additional operators that ensure that RDF value objects are materialized before they are used.

The query plan is submitted to the vectored query engine for execution. The query engine supports both scale-up and scale-out evaluation. For scale-out, operators carry additional annotations that indicate whether they:

- Must run at the query controller (where the query was submitted for execution);
- Must be mapped against the index partition on which the access path will read (for joins)[24]; and
- Can run on any data service in the federation.

The last operator in the query plan writes onto a sink that is drained by the client submitting the query. For scale-out, an operator is added at the end of the query plan to ensure that solutions are copied back to the query controller, where they are accessible to the client. For all other operators, the intermediate solutions are placed onto a work queue for the target operator. The query engine manages the per-operator work queues, schedules the execution of operators, and manages the movement of data on a federation. The full query evaluation sequence is shown in Figure 2.

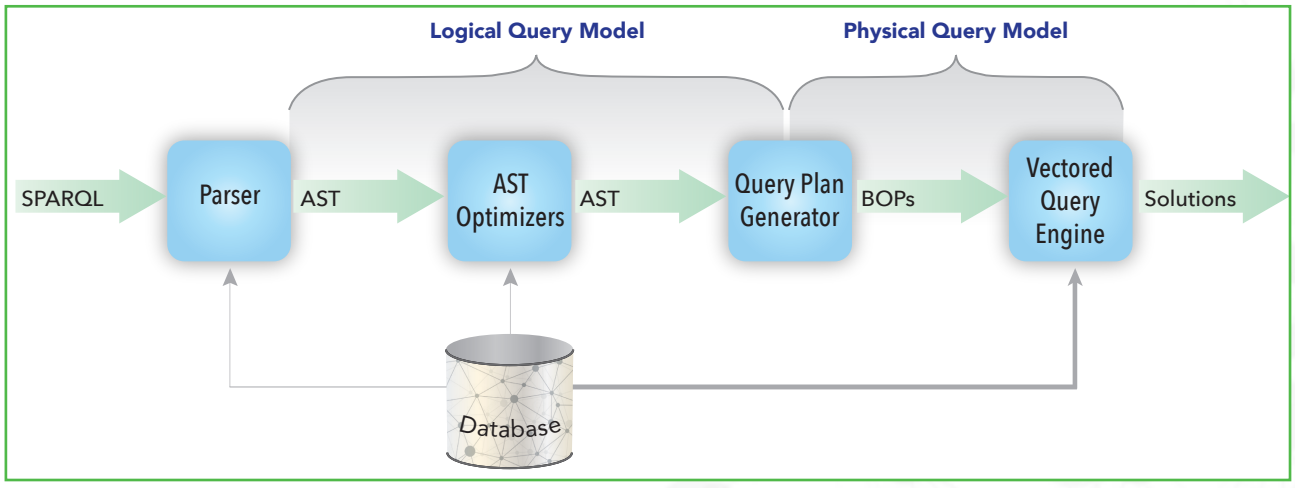


Figure 2: Query execution.

[24] Support for parallel hash joins is planned.

The query engine supports concurrency at several levels:

- Concurrent execution queries. A thread pool in the SPARQL end point controls the number of queries that may execute concurrently.
- Concurrent execution of different operators within the same query. Parallelism here is not limited to avoid the potential for deadlock. Parallelism at this level also helps to ensure that the work queue for each operator remains full and serves to minimize the latency for the query.
- Concurrent execution of the same operator within the same query on different chunks of data. An annotation is used to restrict parallelism at this level.

Solutions are vectored into each operator. Some operators are *"at-once"* and will buffer all intermediate solutions before execution. For example, when evaluating a complex optional, we will fully buffer the intermediate solutions on a hash index before running the sub-group. Other operators are *"blocked"* – they will buffer large blocks of data on the native heap in order to operate on as much data as possible each time they execute – for example, a hash join against an access path scan. However, many operators are "pipelined" – they will execute for each chunk of intermediate solutions.

The Blazegraph Database favors *pipelined* operator execution whenever possible. SPARQL queries involving a sequence of triple patterns are translated using nested index joins and have very low latency to the first solution. Each access path is constrained as solutions flow through the query engine. The constrained access paths are probed using the bindings for each intermediate solution. This turns into highly localized reads on the B+Tree index for that access path. Pipelined execution also is supported for DISTINCT and for simple OPTIONALs (an OPTIONAL containing a single triple pattern and no filters that require materialization of variable bindings against the lexicon).

Query plans involving GROUP BY, ORDER BY, complex OPTIONALs, EXISTS, NOT EXISTS, MINUS, SERVICE, or sub-SELECT have stages that cannot produce any outputs until all solutions have been computed up to that point in the query plan. Such queries can still have low latency as long as the data volume is low. If you want to aggregate or order a subset of the data, then you can move part of the query into a sub-SELECT with a LIMIT, but leave the aggregation or order by clause in the parent query. The sub-SELECT will be pipelined and evaluation will halt as soon as the limit is satisfied. The parent query can then aggregate or order just the data from the sub-SELECT.

Query plans involving sub-GROUPs (including complex OPTIONALs, MINUS, and SERVICE), negation in filters (EXISTS, NOT EXISTS), or sub-SELECTs are all handled in a similar fashion. In each case, an operator that builds a hash index accumulates the intermediate solutions. Once all intermediate solutions have been accumulated, the bindings for the in-scope variables are vectored into the sub-plan. The output solutions from the sub-plan are then joined back against the hash index and vectored into

the remainder of the parent query plan. UNION is handled with a TEE operator. The solutions for each side of the union are vectored into segments of the query plan that execute concurrently.

SPARQL addresses what is in many ways the "easy" problem for graphs – crisp pattern matching against attributed graphs. OPTIONAL adds some flexibility to these graph pattern matches, but does not change the fundamental problem addressed by SPARQL.

We have been tracking with interest current research on heuristic query optimization[25], techniques to counteract latency in distributed query (symmetric hash joins[26] and eddies[27]) and query against open web[28, 29], including frameworks with the potential to support critical thinking about data on the open web[30], including reasoning about evidence supporting conflicting conclusions, unreliable conclusions, and conclusions relying on incomplete evidence[31, 32, 33, 34]. Another line of research on *schema agnostic query*[35] explores *how* people can ask questions about data, especially large and potentially unbounded collections of linked data, when they have little or no a priori understanding of what may be found the data. Similar concerns are also studied as *graph search*[36]. *Graph mining* is concerned with discovering, identifying, aggregating, and summarizing interesting patterns in graphs. As in schema agnostic query, people trying to find interesting patterns in the data often do not know in advance which patterns will be "interesting." Graph mining algorithms can often be expressed as functional vertex programs[37, 38, 39, 40] using multiple full traversals over the graph, and decomposed over parallel hardware.

[25] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. 2012. Heuristics-based query optimisation for SPARQL. In Proceedings of the 15th International Conference on Extending Database Technology (EDBT '12), Elke Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari (Eds.). ACM, New York, NY, USA, 324-335.

[26] Acosta, Maribel, et al. "ANAPSID: AN Adaptive query ProcesSing engIne for sparql enDpoints." The Semantic Web–ISWC 2011 (2011): 18-34.

[27] Avnur, Ron, and Joseph M. Hellerstein. "Eddies: Continuously adaptive query processing." ACM SIGMoD Record 29.2 (2000): 261-272.

[28] Hartig, Olaf, and Johann-Christoph Freytag. "Foundations of traversal based query execution over linked data." Proceedings of the 23rd ACM conference on Hypertext and social media. ACM, 2012.

[29] Theobald, Martin, et al. URDF: Efficient reasoning in uncertain RDF knowledge bases with soft and hard rules. Tech. Rep. MPI-I-2010-5-002, Max Planck Institute Informatics (MPI-INF), 2010.

[30] Günter Ladwig, Thanh Tran, Linked data query processing strategies, Proceedings of the 9th international semantic web conference on The semantic web, November 07-11, 2010, Shanghai, China

[31] Cohen, Marvin S., Freeman, Jared T. and Wolf, Steve. (1996). Meta-recognition in time-stressed decision making: Recognizing, critiquing, and correcting. Journal of the Human Factors and Ergonomics Society (38,2), pp. 206-219.

[32] Cohen, M.S., Thompson, B.B.,Adelman, L., Bresnick, T.A. Lokendra Shastri, & Riedel (2000). Training Critical Thinking for The Battlefield. Volume I: Basis in Cognitive Theory and Research Arlington, VA: Cognitive Technologies, Inc.

[33] Cohen, M.S., Thompson, B.B.,Adelman, L., Bresnick, T.A. Lokendra Shastri, & Riedel (2000). Training Critical Thinking for The Battlefield. Volume III: Modeling and Simulation of Battlefield Critical Thinking. Arlington, VA: Cognitive Technologies, Inc.

[34] Thompson, B.B. & Cohen, M.S. (1999). Naturalistic Decision Making and Models of Computational Intelligence. In Jagota, A. Plate, T., Shastri, L., & Sun, R. (Eds). Connectionist symbol processing: Dead or alive? Neural Computing Surveys 2, pp. 26-28.

[35] Usability of Keyword-Driven Schema Agnostic Search, Lecture Notes in Computer Science Springer Berlin Heidelberg, 2012.

[36] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-Based Approach. SIGMOD, 2004

[37] Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing."Proceedings of the 2010 international conference on Management of data. ACM, 2010.

[38] Low, Yucheng, et al. "Graphlab: A new framework for parallel machine learning." arXiv preprint arXiv:1006.4990 (2010).

[39] Stutz, Philip, Abraham Bernstein, and William Cohen. "Signal/collect: Graph algorithms for the (semantic) web." The Semantic Web–ISWC 2010 (2010): 764-780.

[40] Kyrola, Aapo, Guy Blelloch, and Carlos Guestrin. "GraphChi: Large-scale graph computation on just a PC." OSDI, 2012.

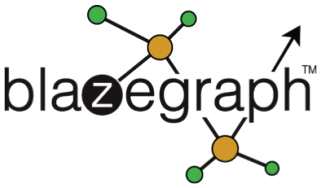
Conclusion

As data volumes explode and organizations face challenges with uncovering insights from multiple data streams, traditional SQL data structures are not adequate for researchers and data scientists who need to explore huge data sets with complex dependencies.

Modern graph databases offer a powerful and efficient way to represent diverse entities and the relationships between them, but in-memory (cache) analytical techniques of popular graph databases are nearly incapacitated as the size of the data sets and relationships between them increase exponentially.

An ultra-scalable, high-performance graph database, Blazegraph addresses the challenges of scaling graphics with the ability to support up to 50 billion edges on a single machine. Blazegraph Database is a proven solution, in use at several Fortune 500 companies, government agencies and other organizations, including AutoDesk, DARPA, EMC, Wikimedia Foundation and Yahoo7.

In our next white paper, we'll take closer look at the Blazegraph Database's scale-up and scale-out architectures.



www.blazegraph.com | blazegraph@ blazegraph.com