Introduction

Bigdata® is a standards-based, high-performance, scalable, open-source graph database. Written entirely in Java, the platform supports the SPARQL 1.1 family of specifications, including Query, Update, Basic Federated Query, and Service Description. Bigdata supports novel extensions for durable named solution sets, efficient storage and querying of reified statement models, and scalable graph analytics. The database supports multi-tenancy and can be deployed as an embedded database, a standalone server, a highly available replication cluster, and as a horizontally-sharded federation of services similar to Google¹s bigtable, Apache Accumulo, or Cassandra.

The bigdata open source platform has been under continuous development since 2006. It is available under a dual licensing model (GPLv2 and commercial licensing) and a number of well-known companies OEM, resell, or embed bigdata in their applications. SYSTAP, LLC leads the development of the open-source project and offers support subscriptions for both commercial and open-source users. Our goal is a robust, scalable, high-performance, and innovative platform. In this whitepaper, we will present the scale-up and scale-out architectures of the bigdata database, index management, and query processing and special concerns for memory management under Java.

Bigdata Database Architecture

In fields as diverse as pharmacology, finance, fraud detection, and intelligence analysis, better analysis and decision making can be facilitated by taking into consideration large amounts of heterogeneous data from many sources in many formats, and degrees of structure, and update rates. Pouring this data together often yields new insights and interesting cross-connections not readily apparent when considering the various data sets in isolation. Such “mash-ups” can provide the basis for operational decision making in complex and dynamic domains, support new forms of online collaboration, and help manage risks in complex markets.

In order to address this problem, we require three things. First, we must be able to load and query very large data sets that exceed the reasonable processing capabilities of even high-end server platforms. Second, those data sets are heterogeneous and interesting data often appears after the system has been deployed, so we must be able to dynamically align the schema for those data sets and to continuously integrate new data. Third, we require the ability to maintain data provenance and drill down into the source detail.

The relational model benefits tremendously from its structure, but lacks the flexibility to rapidly and declaratively integrate new schema into existing systems – relational data integration efforts are often measured in months, not minutes. Expressive Semantic Web technologies such as RDF and OWL have helped reshape this problem, but RDF database technology has not been able to keep up with scale demands. Until very recently, RDF databases and OWL reasoners have not tried to tackle the issues associated with large dynamic data sets, and were insufficiently scalable to attack real world problems where data size can be on the order of billions or even trillions of triples. Without the ability to reach scale, potential Semantic Web adopters turn to cloud computing technologies such as map/reduce, not fully understanding the tradeoffs between the two technologies and, in particular, the limitations of map/reduce processing for handling graph structured or linked data.

Bigdata® [1] [2] 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, bigdata uses dynamically partitioned key-range shards. This architecture was chosen to remove any realistic scaling limits – in principle, bigdata may be deployed on 10s, 100s, or even thousands of machines. Further, and unlike hash-partitioned approaches, new capacity may be added incrementally to data centers without requiring the full reload of all data. On top of that core is the bigdata RDF Store, a massively scalable RDF database supporting RDFS and OWL Lite reasoning, high-level query (SPARQL), and datum level provenance.

Deployment models

Bigdata supports several distinct deployment models:

- Embedded Database (Journal)
- Servlet Engine (Journal in WAR)
- Replication Cluster (HA Journal)
- Horizontally scaled, parallel database (Federation) aka scale-out.

These deployment models are based on two distinct architectures. The embedded database, WAR, and the replication cluster are *scale-up* architectures based on the Journal. The Journal provides basic support for index management against a single backing store. The Federation is a *scale-out* architecture using dynamically partitioned indices to distribute the data within each index across the resources of a compute cluster.

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. Also, while updates on the Journal and replication cluster are ACID, updates on the federation are *shard-wise* ACID. Finally, while it is always important to vector operations against indices, but vectored operations are absolutely required for good performance on the scale-out architecture.

The choice of the right deployment model depends on your requirements. The Journal offers low latency operations due to its *locality* and 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 due to the overhead of inter-node coordination and offers greater throughput for some query workloads. The federation can scale-out far beyond the Journal, but due to higher coordination costs, you need at least 3 machines to have performance similar to a single machine Journal. Since you can have the low-latency of the Journal combined with high availability and horizontally scaled query on a 3-node replication cluster, the scale-out architecture of the federation really only makes sense for very large data sets and clusters of 8 or more machines.

All deployment models support the SAIL, SPARQL 1.1 Query, SPARQL Update, etc.

---

[1] http://www.bigdata.com/blog

[2] http://www.sourceforge.net/projects/bigdata

Concurrency Control

Bigdata supports optional transactions based on MVCC. 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 writes 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 allows 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 (in fact, bigdata can resolve many 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 their "row store" [3] to address a set of very specific application requirements. In particular, they 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 their design, Google opted to restrict concurrency control to ACID[4] operations on "rows" within a "column family." With this design, a purely local locking scheme may be used and substantially higher concurrency may be obtained. Bigdata uses this approach for its "row store", for the lexicon for an RDF database, and for high throughput distributed bulk data load.

For a federation, distributed transactions[5] are primarily used to support snapshot isolation for query. An "isolatable" index (one which 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 as discussed below.

Managing database history

Bigdata is 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, essentially winding back the clock to the state of the database at some prior

---

[3] "Bigtable: A Distributed Storage System for Structured Data",http://labs.google.com/papers/bigtable.html

[4] 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.

[5] Bigdata® 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.

day, month or year. This feature can be used in many interesting ways, including regulatory compliance, examining changes in the state of accounts over time, etc.

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. This is done by specifying the minimum age before a commit point may be released, e.g., 5 minutes, 1 day, 2 weeks, or 12 months. The minimum release age can also be set to zero, in which case bigdata 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 5 day history retention policy, and you insert a tuple into an index, then that tuple would remain in the index until 5 days after it was *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 5 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, update in logarithmic amortized time. The bigdata 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. 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. bigdata uses a constant (and configurable) branching factor and allows the page size of the index to vary. This works out well with overall copy-on-write architecture and simplifies some decisions in the maintenance of the index.

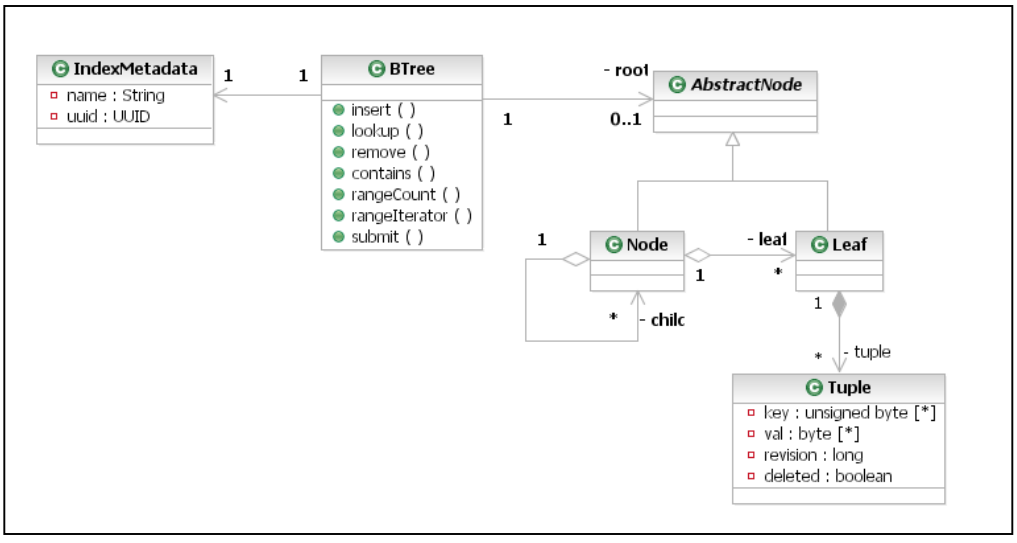


Figure 1 -- B+Tree architecture.

In bigdata, an index maps *unsigned* byte[] keys to byte[] values[6]. Mechanisms are provided which 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 which 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 Multi-Version Concurrency Control (MVCC)[7]. 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[8] [9] 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), methods which operate on key ranges (rangeCount, rangeIterator), and a set of methods to submit Java procedures that are mapped against the index and execute locally on the appropriate data services (see below). 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 #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 are also available on a federation, where a key-range may span multiple index partitions.

Scale-Up Architecture

The Journal manages a backing store, provides low-level mechanisms for writing and reading allocations on that file, and has higher-level mechanisms for registering and operating on indices. There are several different backing store models for the Journal. The most important are described below.

---

[6] We are reviewing this design decision with respect to column-wise storage.

[7] Reed, D.P.. "Naming and Synchronization in a Decentralized Computer System". *MIT dissertation*. http://www.lcs.mit.edu/publications/specpub.php?id=773

[8] Huffman coding, http://en.wikipedia.org/wiki/Huffman_coding

[9] Canonical Huffman coding, http://en.wikipedia.org/wiki/Canonical_Huffman_code

WORM

The WORM is a Write Once Read Many store. It is an *indelible* append-only file structure, with root blocks that are updated at each commit point. The WORM is primarily used to buffer writes in the scale-out architecture before they are migrated onto read-optimized, read-only B+Tree files.

RWStore

The RWStore provides a read/write model based on managed allocation slots on the backing file and can address up to 16TB of data. The vast majority of the allocations are the nodes and leaves of the indices. As noted above, index updates use a *copy-on-write* model. The old version of the index page is deleted, but it will remain visible until (a) no open transaction is reading on a commit point in which that index page is visible; and (b) the history retention period has expired for commit points in which the page is visible. These criteria are summarized and tracked as the earliest release time. Commit points *before* that release time may be released and their allocations recycled. The recycler does not use a vacuum process. Instead, the addresses of the deleted pages are written onto delete blocks. When the commit point is released, the delete blocks are read and the associated pages are bit flagged as free in the allocators.

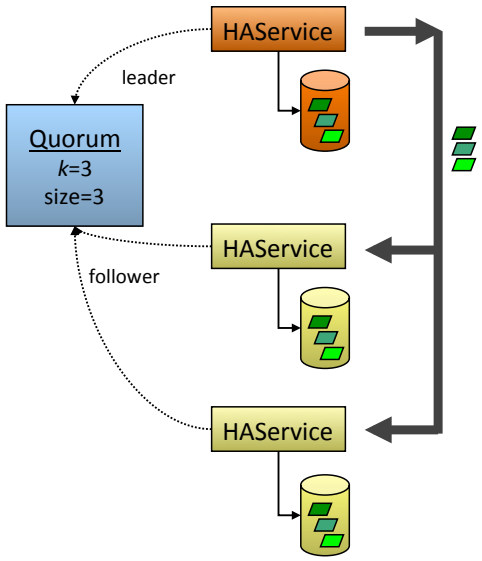
MemStore

The MemStore provides a similar capability for managed allocations, but the data are stored in the C heap (rather than the Java managed object heap). This avoids problems associated with garbage collection overhead for high object creation / retention rates. The MemStore is 100% Java. It relies on NIO buffers to create allocations outside of the Java managed object heap. The MemStore is used internally in combination with the HTree[10] data structure for analytic query operations requiring highly scalable hash indices.

High Availability

The HAJournalServer provides a highly available replication cluster for the scale-up database deployment architecture (the Journal). The HAJournalServer provides horizontal scaling of query, not data. Since the data is fully replicated on each node, query scales linearly in the size of the replication cluster. Because query relies entirely on local indices, the HAJournalServer offers the same low latency for query as the Journal. In contrast, in a scale-out deployment, the data is partitioned and distributed across the nodes of a cluster. Because of this partitioning, query evaluation must be coordinated across multiple nodes in a scale-out deployment. Due to this higher coordination overhead, scale-out query has higher latency, but can achieve higher throughput by doing more parallel work. Due to the combination of low latency query, horizontal scaling of query, and high availability, the HAJournalServer deployment model should be preferred when the data scale will be less than 50 billions and when low latency for individual queries is more important than throughput on high data volume queries.

[10] A Robust Scheme for Multilevel Extendible Hashing by Sven Helmer, Thomas Neumann, Guido Moerkotte. ISCIS 2003: 220-227



High availability is based on a quorum model and the low-level replication of write cache blocks across a pipeline of services. A highly available service exposes an RMI interface using Apache River and establishes watchers (that reflect) and actors (that influence) the distributed quorum state in Apache zookeeper. Sockets are used for efficient transfer of write cache blocks along the write pipeline. The services publish themselves through zookeeper. Services register with the quorum for a given logical service. A majority of services must form a consensus around the last commit point on the database. One of those services is elected as the leader and the others are elected as followers (collectively, these are referred to as the joined services – the services that are joined with the met quorum). Once a quorum meets, the leader services write requests while reads may be served by the leader or any of the followers. The followers are fully consistent with the leader at each commit point. If a follower can not commit, it will drop out of the quorum and resynchronize before re-entering the quorum.

Write replication occurs at the level of 1MB cache blocks. Each cache blocks typically contain many records, as well as indicating records that have been released. Writes are coalesced in the cache on the leader, leading to a very significant reduction in disk and network IO. Followers receive and relay write cache blocks and also lay them down on the local backing store. In addition, both the leaders and the followers write the cache blocks onto a HALog file. The write pipeline is flushed before each commit to ensure that all services are synchronized at each commit point. A 2-phase commit protocol is used. If a majority of the joined services votes for a commit, then the root blocks are applied. Otherwise the write set is discarded. This provides an ACID guarantee for the highly available replication cluster.

HALog files play an important role in the HA architecture. Each HALog file contains the entire write set for a commit point, together with the opening and closing of root blocks for that commit point. HALog files provide the basis for both incremental backup, online resynchronization of services after a temporary disconnect, and online disaster recovery of a service from the other services in a quorum. HALog files are retained until the later of (a) their capture by an online backup mechanism, and (b) a fully met quorum.

Online resynchronization is achieved by replaying the HALog files from the leader for the missing commit points. The service will go through a local commit point for each HALog file it replays. Once it catches up it will join the already met quorum. If any HALog files are unavailable or corrupt, then an online rebuild replicates the leader's committed state and then enters the resynchronization protocol. These processes are automatic.

Online backup uses the same mechanisms. Incremental backups request any new HALog files, and write them into a locally accessible directory. Full backups request a copy of the leader's backing store. The replication cluster remains online during backups. Restore is an offline process. The desired full backup and any subsequent HALog files are copied into the data directory of the service. When the service starts, it will apply all HALog files for commit points more recent than the last commit point on the Journal. Once the HALog files have been

replayed, the service will seek a consensus (if no quorum is met) or attempt to resynchronize and join an already met quorum.

Scale-Out Architecture

Bigdata is a general-purpose, horizontally scaled architecture for persistent, ordered data. The overall approach utilizes key-range partitioned B+Tree[11] indices distributed across the resources of a cluster. This choice was influenced by previous work on distributed database architectures, including Google's bigtable[12] project.

Services Architecture

The bigdata federation is a services architecture. The *Data Services* correspond to the concept of a tablet server in Google's bigtable or Apache Accumulo. Each Data Service has responsibility for some number of index partitions. However, unlike those platforms, the Data Service can support distributed query processing as well as servicing read and write requests. This makes it possible to co-locate JOIN processing with the data on which a JOIN must read. The *Metadata Service* is also referred to as the *shard locator service* – it maintains a B+Tree over the key-range partitions for each scale-out index, mapping each key-range partition onto a partition metadata record required to locate the index partition. Clients requiring a key-range scan of an index will obtain a *locator* scan for that key-range. The locator scan visits the partition metadata records. The client then issues separate requests to the data service for each index partition. A transaction service coordinates read locks to support snapshot isolation across the cluster and tracks the earliest commit point that must be retained by the data services in order to satisfy the open transactions and the configured history retention period for the database. Client services provide a container for executing distributed tasks. Jini (now the Apache river project) is used for service registry and discovery. Global synchronous locks and configuration management are realized using Apache zookeeper. Support for SPARQL processing is achieved by integration with the Sesame 2 platform[13].

---

[11] Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1 (1972) 173-189.; Bayer, R. and Unterauer, Prefix B-trees. *ACM Trans. On Database Systems*, 2,1 (Mar., 1977) 11-26

[12] "Bigtable: A Distributed Storage System for Structured Data",http://labs.google.com/papers/bigtable.html

[13] Support for additional RDF platforms, including Jena, is being considered.

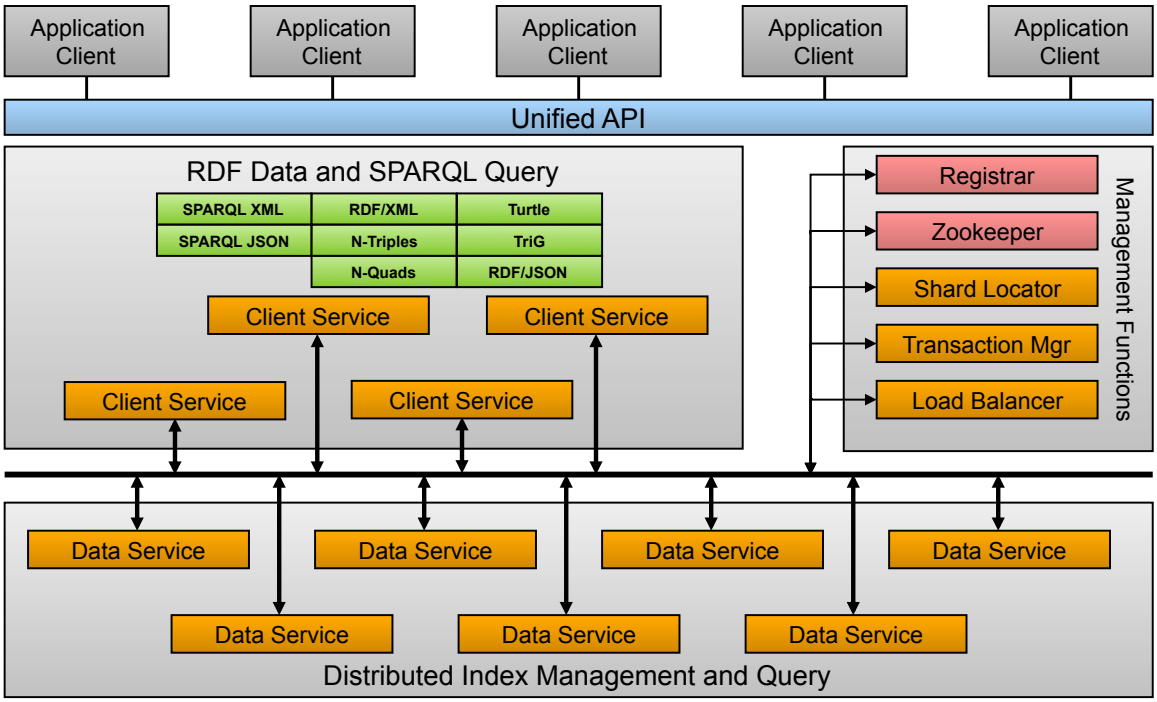


Figure 2 The services architecture for a bigdata federation.

Service Discovery

Bigdata services advertise themselves in a jini service registrar and are discovered by lookups against that registrar. Clients await discovery of the transaction service and the metadata service, and then register or lookup indices using the metadata service. The metadata service maps key ranges for each scale-out index onto logical data services. When a client issues a request against a scale-out index, the bigdata library transparently resolves the locator(s) for that query. Clients obtain proxies for the data services using jini, then talk directly to the data services. This process is illustrated in Figure 3, and is completely transparent to bigdata applications. The client library automatically handles redirects when an index partition is moved, split or joins and data service failover.

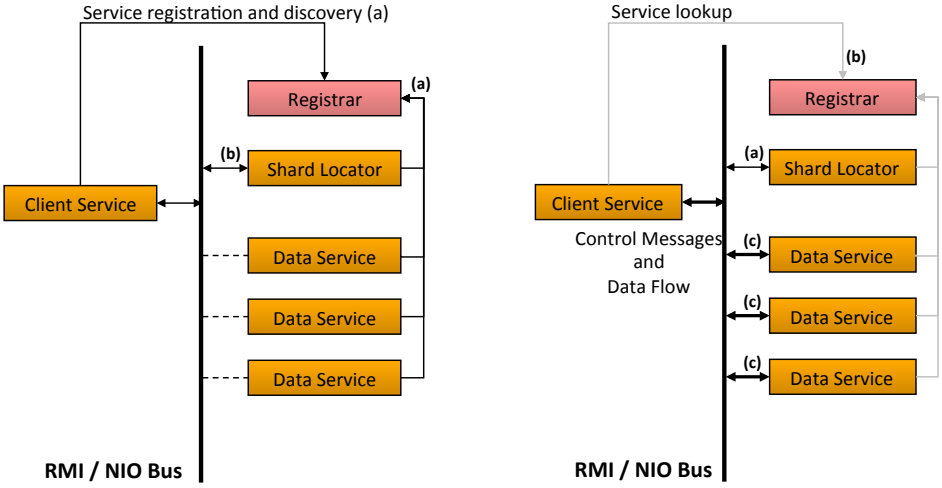


Figure 3: Service discovery.

Left: Clients and services advertise themselves with the service registrar (a). Clients discover the shard locator (b).

Right: Clients discover the locations of index shards (a), discover the data services hosting those shards (b), and then talk directly to those data services (c).

Dynamic Partitioning

Bigdata indices are dynamically broken down into key-range shards, called *index partitions*, in a manner that is completely transparent to clients. Each index partition is a collection of local resources that contain all tuples for some key-range of a scale-out index and is assigned a unique identifier (a 32-bit integer).

There are three basic operations on an index partition: *split*, which divides an index partition into two (or more) index partitions covering the same key-range; *move*, which moves an index partition from one data service to another, typically on a different machine in the cluster; and *join*, which joins two index partitions that are siblings, creating a single index partition covering the same key-range. These operations are invoked transparently and asynchronously.

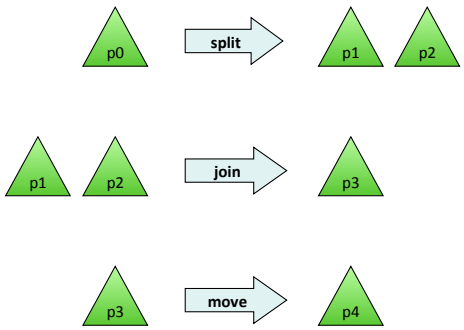


Figure 4 Basic operations on index partitions.

The data in the indices is strictly conserved by these operations, only the index partition *identifier*, the index partition *boundaries* (split, join), or the index partition *location* (move) are changed. The index partition identifier is linked to a specific key-range and a specific location. Since these operations change either the key-range and/or the location, they always assign a new index partition identifier. Requests for old index partitions are easily recognized as having index partition identifiers that have been retired and result in *stale locator exceptions*. The client-side views of the scale-out indices automatically trap stale locator exceptions and redirect and reissue requests as necessary.

Metadata Service

Index partition *locators* are maintained in a *metadata index* that lives on a specialized *data service* known as the *metadata service*. An index partition locator maps a key-range for an index onto an index partition identifier and the data service hosting that index partition. The key for the tuples in the metadata index is simply the first key that could enter the corresponding index partition. Depending on the data scale, there may be thousands of index partitions per scale-out index.

Data Services

Each data service maintains an append-only write buffer (a WORM mode Journal) and an arbitrary number of read-only, read-optimized *index segments*. Each index partition is, in fact, a *view* onto (a) the mutable B+Tree on the live journal; and (b) historical data on a combination of

old journals and index segments. The nominal capacity of the Data Service journal is ~200M. Likewise, the target size for the index segments in a compact index partition view is ~200M. There may be 100s or 1000s of index partitions per data service. Thus index segment files form the vast majority of the persistent state managed by a data service.

Index Segments

Index segments have been briefly discussed above. Each index segment is the result of a batch build operation and has data for some key range of an index as of some commit point on the database. The index segment is optimized for read performance. The nodes of the B+Tree are laid out in key order on the disk and are typically read in a single IO when the index segment is opened. The leaves are also laid out in key-order on the disk and are linked to both their predecessors and followers in key order. A single IO is required to read a leaf from the disk, and sequential scans can be performed efficient in either direction.

Bloom filters

A *bloom filter* is an in memory data structure that can very rapidly determine whether a key IS NOT in an index. When the bloom filter reports "no", you are done and you do not touch the index. When the bloom filter reports "yes", you have to read the index to verify that there really is a hit. Bloom filters are a stochastic data structure, require about 1 byte per index entry, and must be provisioned up front for an expected number of index entries. So if you expect 10M triples that is a 10MB data structure. Since bloom filters do not scale-up, they are automatically disabled once the number of index entries in the mutable B+Tree exceeds about 2M tuples.

Bloom filters may be configured for scale-out indices. Each time an index partition build or merge operation generates a new index segment file, the data on the mutable B+Tree is migrated into read-optimized index segments. Every time we overflow a journal, we wind up with a new (empty) B+Tree to absorb writes, so the bloom filter on the journal is automatically re-enabled. Further, during build and merge operations we have perfect knowledge of the number of tuples in an index segment and generate an exact fit bloom filter. This can provide a dramatic boost when a distributed query includes joins that wind up doing a large number of point lookups to verify that a fully bound triple pattern exists in the data.

Overflow Processing

Periodically writes on a data service cause the journal to reach its nominal size on the disk – this is known as an “overflow.” When this occurs, a new journal is created, and an asynchronous process begins which migrates buffered writes from the old journal onto new index segments. Asynchronous overflow processing defines two additional operations on index partitions: *build*, which copies *only* the buffered writes for the index partitions from the old journal onto a new index segment; and *compacting merge*, which copies all tuples in the index partition view into a new index segment. Index partition *builds* make it possible to quickly retire the old journal, but they add a requirement to maintain delete markers on tuples in order to prevent historical tuples from re-entering the index partition view. Index partition *merges* are more expensive, but they produce a compact view of the tuples in which duplicates have been eradicated. The decision to *build* vs. *merge* is made locally based on the complexity of the index partition view and the relative requirements of different index views for a data service. The asynchronous overflow tasks are arranged in a priority queue. Separate thread pools are used to limit the number of concurrent build tasks and concurrent merge tasks. The decision to *split* an index partition into two index partitions or to *join* two index partitions into a single index partition is made after a *merge* when there is a good estimate of the space requirements on disk

for the index partition. The decision to move an index partition is based on load. A *merge* is always performed before a move to produce a compact view that is then sent across a socket to the receiving service[14].

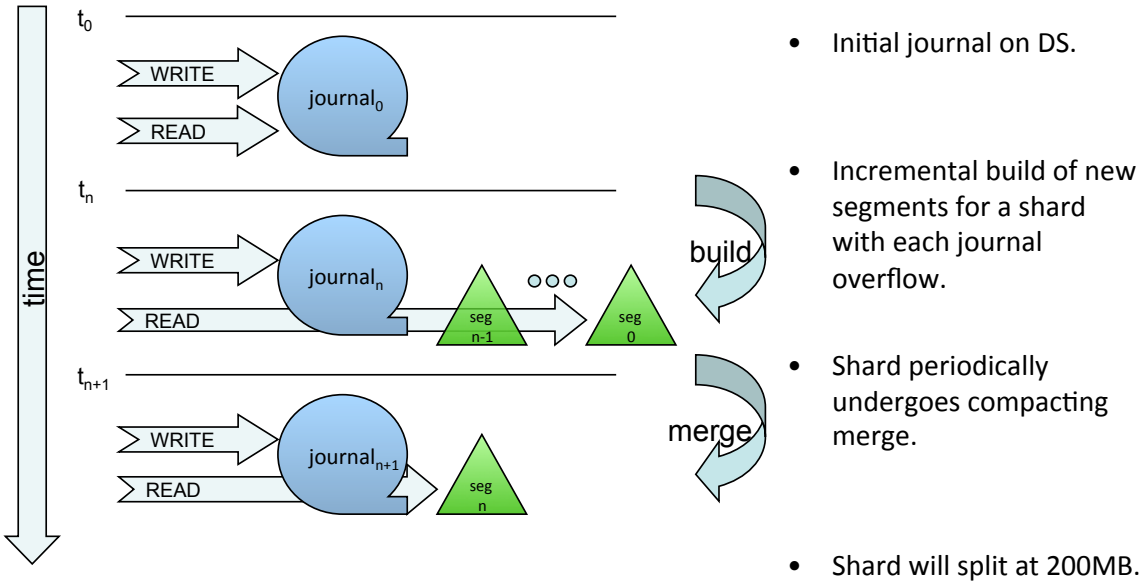


Figure 5: Diagram illustrating how the view of a shard evolves over time. The index segment files represent data from the previous journal. A new journal is opened each time the current journal files up.

When a scale-out index is registered, the following actions are taken: First, a metadata index is created for that scale-out index on the metadata service. This will be used to locate the index partitions for that scale-out index. Second, a single index partition is created an arbitrary data service. Third, a locator is inserted into the metadata index mapping the key-range ([],∞) onto that index partition[15]. Clients resolve the metadata service, and probe it to obtain the locator(s) for the desired scale-out index. The locator contains the data service identifier as well as the key-range ([],∞) for the index partition. Clients then resolve the data service identifier to the data service and begin writing on the index partition on that data service.

[14] A similar design was described in "Bigtable: A Distributed Storage System for Structured Data", http://labs.google.com/papers/bigtable.html.

[15] All keys are translated into unsigned byte[]s. An empty byte[] is the first possible key in any bigdata index. The symbol ∞ is used to indicate an arbitrarily long unsigned byte[] containing 0xFF in all positions and corresponds to the greatest possible key in any bigdata index and is indicated internally by a *null* reference.

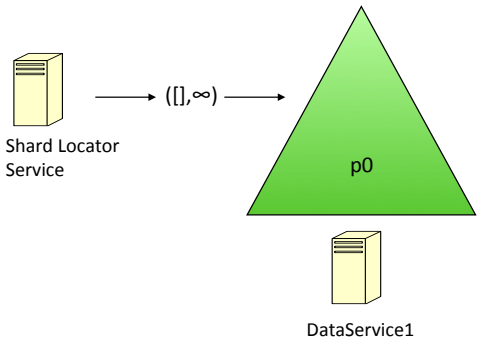


Figure 6 Initial conditions place a single index partition on an arbitrary host. That index partition contains all data for the scale-out index.

Eventually, writes on the initial index partition will cause its size on disk to exceed a configured threshold (~200M) and the index partition will be split. The split(s) are identified by examining the tuples in the index partition and choosing one or more *separator key(s)*. Each separator key specifies the first key which may enter a given index partition. The separator keys for the locators of a scale-out index always span the key range ([],∞) without overlap. Thus each key always falls into precisely one index partition.

If necessary, applications may place additional constraints on the choice of the separator key. For example, this may be done to ensure that an index partition never splits a logical row. That guarantee may be used to achieve extremely high concurrent write rates using shard-wise ACID operations since concurrency control may be conducted locally on the data service.

Scatter Split

The potential throughput of an index increases as it is split and distributed across the machines in the cluster. In order to rapidly distribute an index across the cluster and thereby increase the resources that can be brought to bear on that index, a *scatter split* is performed early in the life cycle of the first index partition for each scale-out index.

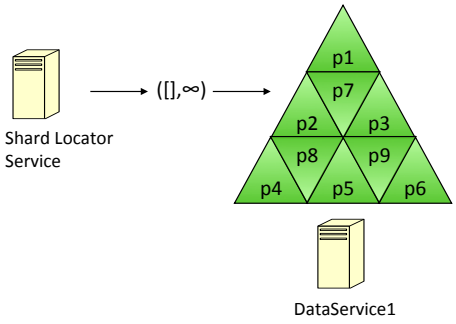


Figure 7 Preparing for the initial scatter-split of an index.

Unlike a normal split, which replaces one index partition with two index partitions, the scatter split replaces the initial index partition with N * M index partitions, where N is the number of data services and M is a small integer.

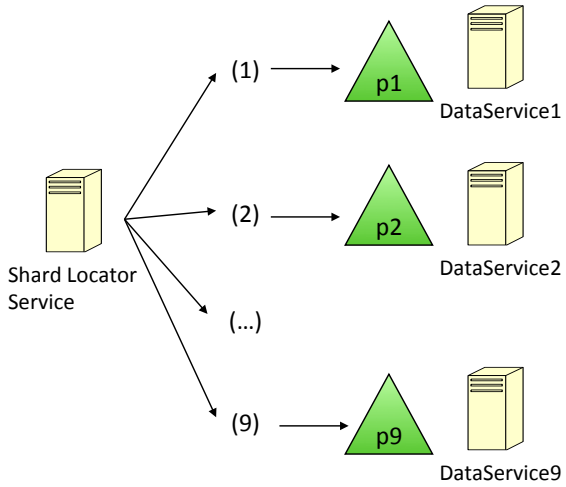


Figure 8 After the scatter split. The index has been distributed across the resources of the cluster. In this example, there are nine data services in the cluster. There can be hundreds.

The new index partitions are redistributed across the cluster, leaving every N^{th} index partition on the original data service. After the scatter-split operation, the throughput of the index may be dramatically increased.

RDF Database Architecture

In this section we define the Resource Description Framework (RDF), and show how an RDF database is realized using the bigdata architecture. Bigdata implements the Storage And Inference Layer (SAIL) API, which provides a pluggable backend for the Sesame platform [16]. However, the query evaluation and transaction models for bigdata differ significantly from those of openrdf.

Resource Description Framework

The Resource Description Framework[17] [18] (RDF) 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 (our 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 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[19], whereas Literal values describe constants such as character strings and

[16] Openrdf, http://www.openrdf.org

[17] Resource Description Framework, http://www.w3.org/RDF/

[18] RDF Semantics, http://www.w3.org/TR/rdf-mt/

[19] The benefit of URIs over traditional identifiers is two fold. First, by using URIs, RDF may be 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.

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[20]) 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[21] and the OWL Ontology Web Language[22] (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 only gets more so as data size increases. Bigdata has focused on scale over expressivity.

Database Schema for the RDF

Bigdata supports three distinct RDF database modes: triples, triples with provenance[23], 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 bigdata 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. Bigdata 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 are use a variable length encoding. URIs declared in a vocabulary when the KB instance was created are also inlined (in 2-3 bytes). Depending on the configuration, blank nodes are typically inlined. As discussed elsewhere, 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

[20] http://www.w3.org/TR/rdf-syntax-grammar/

[21] http://www.w3.org/TR/rdf-schema/

[22] http://www.w3.org/2004/OWL/

[23] We are in the process of reconciling our statement level provenance mode with efficient support for RDF reification. When this process is finished, we will support efficient statements about statement in both the triples and quads modes of the database.

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:

- 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 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. This allows 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 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 [24]. 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. For a triple store, this requires 3 statement indices (SPO, POS, and OSP). For a quad store, this requires 6 statement indices (OCSP, SPOC, CSPO, PCSO, POCS, and SPOC). In each case the name of the index indicates the manner in which the **S**ubject, **P**redicate, **O**bject, and the optional **C**ontext 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 SSD to reduce the IO Wait for the disk.

[24] 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.

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 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.

Bigdata 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 which are known to evaluate to a constant, ensuring that query plans are consistent with the bottom-up evaluation semantics of SPARQL, reordering joins, attaching FILTERS in the most advantageous locations, etc. 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 50-100 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 [25], and an analysis of the propagation of in-scope variables between sub-groups and sub-SELECTs.

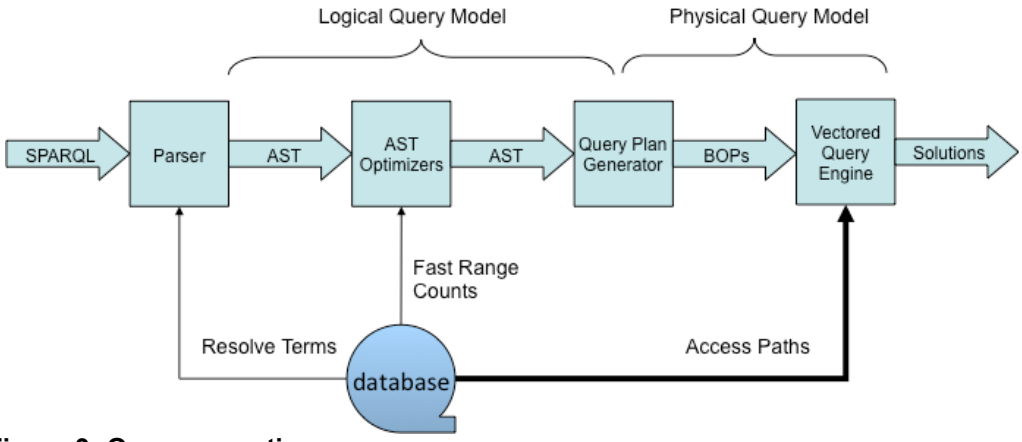


Figure 9: Query execution.

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 may also be optional joins, sub-SELECT joins, joins of pre-computed named solution sets, etc.

[25] A *runtime* join ordering algorithm based on chain sampling has been implemented, but is not yet integrated into the SPARQL query engine.

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 which indicate whether they must run at the query controller (where the query was submitted for execution), whether they must be mapped against the index partition on which the access path will read (for joins) [26], and whether they 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 that submitted the query. For scale-out, an operator in 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.

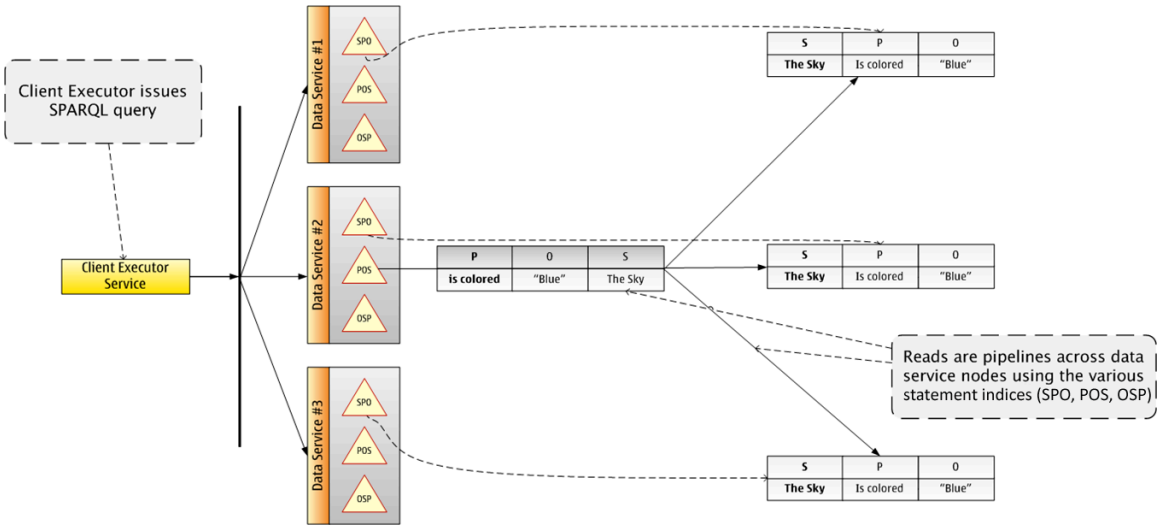


Figure 10: Illustration of pipelined join execution in scale-out.

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

[26] Support for parallel hash joins is planned.

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 the 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.

Bigdata 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 a highly localized reads on the B+Tree index for that access path. Pipelined execution is also 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).

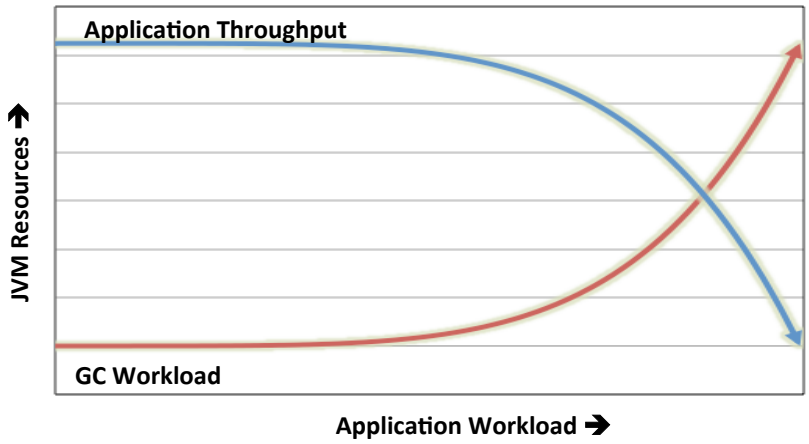
Query plans involving GROUP BY, ORDER BY, complex OPTIONALs, EXISTS, NOT EXISTS, MINUS, SERVICE, or sub-SELECT have stages that cannot produce any outputs until all solutions have been computed up to that point in the query plan. Such queries can still have low latency as long as the data volume is low. 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.

Analytic Query Mode

Incremental compilation and sophisticated runtime hot spot analysis of Java applications often yields code as fast a hand-coded C. However, the managed object heap is a known weak point in the JVM. As the object creation rates and object retention period increase, the duty cycle time of the garbage collector increases. Eventually, the garbage collection begins to lock out the application for significant periods of time [27].

[27] This issue is well recognized in Java cache fabrics and has led to a variety of technologies using the *native* C heap rather than the managed object heap.



To address this problem, bigdata provides two implementations for each operation based on a hash index. One version of the operator uses the JVM collection classes. These classes provide excellent performance for modest collection sizes, and in some cases offer very high concurrency. The other version of the operator is based on the HTree index structure and the MemStore (which is backed by the *native* heap of the C process). These operators scale to very large data sets without any overhead from the garbage collector. For low-latency, select queries the performance of the native memory operators is close to the performance of the JVM versions of the same operator [28]. However, when the queries must materialize large 100s of millions of solutions, the JVM collection classes incur very high GC costs but the native memory operators scale gracefully. When the *analytic* query mode is specified, the query plan will use the native memory operators. The analytic query mode may be enabled by a checkbox on the HTML FORM, a URL query parameter (*?analytic*), or a query hint (*hint:analytic*) may be used to enable the analytic query mode [29].

Inference and truth maintenance

RDF model theory defines various entailments. The entailments are triples *not* explicitly given in the input, but the database must behave *as if* those triples were present. There are broadly speaking two ways of handling such entailments. First, they can be computed up-front when the data are loaded and stored in the database alongside the explicitly given triples. This approach is known as *eager closure* because you compute the closure of the model theory over the explicit triples and materialize those *inferred* triples in the database. The primary advantage of eager closure is that it materializes all data, both explicit and inferred, in the database. This greatly simplifies query planning and provides equally fast access paths for entailed and explicit statements. Eager closure can be extremely efficient, but there can still be significant latency, especially for very large data sets, as the time to compute the closure is often on the order of the time to load the raw data. The other drawback is *space* as the inferred triples are stored in the indices, thereby inflating the on disk size of the data set.

The second approach is to materialize the inferences at query time. This has the advantage that the data set may be queried as soon as the raw data have been loaded and the storage requirements are those for just the raw data. There are a variety of techniques for doing this,

[28] When executing the BSBM benchmark with the 100M triple data sets, the performance using the native memory operators is within 10% of the performance of the JVM based operators. The performance difference is mostly due to the overhead of serialization of the solution sets onto the native memory pages. However, the JVM DISTINCT operator allows more concurrency and can outperform the native memory DISTINCT operator that has to single-thread the updates on the underlying HTree index.

[29] The query hint must be used if you are not using the NanoSparqlServer as the SPARQL end point.

including *backward reasoning* [30] [31], which is often used in Prolog systems, and *magic sets* [32], which is often used in datalog systems. With SPARQL 1.1, property paths can also be used to embed select inferences into queries.

An RDF database that utilizes an *eager closure* strategy faces another concern. It must maintain a coherent state for the database, including the inferred triples, as data are added to or removed from the database. This problem is known as *truth maintenance.* For RDF Schema, truth maintenance is trivial when adding new data. However, it can become quite complex when data are removed, as a search must be conducted to determine whether or not inferences already in the database are still *entailed* without the retracted assertions.

Once again, there are several ways to handle this problem. One extreme is to throw away the inferences, deleting them from the database, and then re-compute the full forward closure of the remaining statements. This has all the drawbacks associated with eager closure and even a trivial retraction can cause the entire closure to be re-computed. Second, truth maintenance can be achieved by storing *proof chains* in an index[33]. When a statement is retracted, the entailments of that statement are computed and, for each such entailment, the proof chains are consulted to determine whether or not the statement is still proven without the retracted assertion. However, storing the proof chains can be cumbersome. Third, magic sets once again offer an efficient alternative for a system using eager closure to pre-materialize inferences. Rather than storing the proof chains, we can simply compute the set of entailments for the statements to be retracted and then submit queries against the database in which we inquire whether or not those statements are still proven.

Bigdata supports a hybrid approach in which the eager closure is taken for some RDF Schema entailments while other entailments are only materialized at query time. This approach is not uncommon among RDF databases. In addition, the scale-up architecture also supports truth maintenance based on storing proof chains. Truth maintenance is not available in scale-out because all updates would have to be serialized (executed in a single thread) in order for truth maintenance to have well defined semantics.

Reification done right

If you have a background with publishing, then you probably think of RDF as a *metadata* standard. RDF statements provide metadata about resources. However, for interesting historical reasons[34], RDF lacks a good solution for *metadata* about *metadata* – what is

---

[30] Robinson, J. A. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (Jan. 1965), 23-41.

[31] Cohen, J. 1996. Logic programming and constraint logic programming. *ACM Comput. Surv.* 28, 1 (Mar. 1996), 257-259.

[32] J. D. Ullman, Bottom-up beats top-down for datalog, Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, p.140-149, March 1989, Philadelphia, Pennsylvania, United States.

[33] J. Broekstra, A. Kampman, Inferencing and Truth Maintenance in RDF Schema : Exploring a naive practical approach, http://www.openrdf.org/doc/papers/inferencing.pdf, in Workshop on Practical and Scalable Semantic Systems (PSSS), 2003.

[34] RDF was shaped by the constraints of first order predicate logic. Allowing statements about statements into RDF model theory shifts that logic from first order predicate calculus, which does not permit statements about statements, into second order predicate calculus. The original concept of the

sometimes called *statements about statements*. There are two mechanisms that may be used to capture provenance: RDF reification and named graphs. Both are sources of confusion. RDF reification creates a model of a statement[35], but does not assert the existence of the statement that it models. Even for the RDF/SPARQL community, reading, writing, and thinking in reified statement models is an awkward and unpleasant business.

For some applications, it is sufficient to know the "source" of the document containing the assertions. In this case, *named graphs* are a good fit – the source is simply the name of the graph. However, for many domains it is critical to know the *provenance* of each assertion, including who, when, where, etc. This amounts to statement level provenance. Some security models also require the ability to specify the permissions for each datum independently. Often these requirements are found in the same systems.

RDF can also been seen as a *graph* standard. The URIs are vertices, the statements are edges or attributes. However, for many graph mining applications, graphs are collections of weighted edges. To this community, statement models look like an extremely complex and ill-suited approach to representing what is, essentially, a sparse matrix. The broader graph database community uses a property graph model[36] where both vertices and edges may have attributes[37]. The RDF syntax obviously handles vertex attributes, but it leaves people scratching their heads when they try to understand how to capture *link attributes*.

RDF needs to be able to support all of these use cases in an efficient and easy to understand manner.

Over the years, we have implemented a number of different mechanisms in the bigdata platform, including statement identifiers that were assigned by the lexicon and representing statements about statements through *inlining*. We are now moving towards a grand synthesis of these concepts – something that we call *reification done right* [38]. The key insights are (1) RDF reification does not dictate the physical schema; (2) RDF reification can be explicitly reconciled with statements about statements in the model theory[39]. This means that we can *automatically* index reified statement models using a physical schema that is significantly more efficient in both space and query performance. It also means, that we known exactly how to translate between a sparse matrix model or a property graph model and RDF; and (3) we need a better syntax – especially for query.

Bigdata has a dedicated triples + provenance database mode. This is based on the notion of statement identifiers[40]. Internally, statement identifiers are represented as variable length IVs

---

Semantic Web steered clear of second order predicate calculus in order to avoid some pitfalls associated with previous knowledge representation frameworks.

[35] RDF Semantics, http://www.w3.org/TR/rdf-mt/

[36] https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model

[37] Edges are often restricted to simple attributes. However, the topic maps data model and hypergraphs both allow edges to double as vertices.

[38] The key insights here are due to a working group formed at the 2012 Dagstuhl Semantic Data Management workshop, with special thanks to Olaf Hartig, Tran Thanh, Orri Erring, and Yrjana Rankka.

[39] Many thanks to Olaf Hartig for this work on this.

[40] Statement identifiers reflect a concern best articulated in Topic Maps as the ability to make assertions about anything, even other assertions. Topic Maps are not less concerned with model theory and entailments and focus more on an architecture for making assertions about subjects, including that two resources may identify the same subject.

whose encoding is precisely the encoding of the nested statement – plus a *flags* byte marking this as a statement identifier. The advantage of this approach is that we do not need to store statement in the lexicon and we can immediately decompose a statement about statements into its component parts.

There are three drawbacks to the current implementation. First, it hijacks the semantics of the GRAPH keyword in SPARQL to bind the statement as a variable and is therefore not compatible with indexing quads. Second, it relies on an extension to the RDF/XML syntax to interchange statements about statements. Third, the inlining technique relies on a *prefix* marker. Therefore, statements about a statement are not co-located with the ground statement in the indices. We plan to fix these issues in a future release, thus allowing efficient indexing and querying of statements about statements to be used transparently in either the triples or quads mode of the database.

The following examples illustrate the how this will work. The double chevrons indicate where one statement is nested within another. Unlike RDF reification, this syntax also implies the existence of the statement within the double chevrons [41]. In the indices, bigdata represents the statement about a statement as the composition of the IVs of its components, including the nested statement[42].

```
<<:SAP :bought :sybase>> dc:source reuters:us-sybase .
```

This same syntax is supported for query:

```
SELECT ?src ?who {
   <<?who :bought :sybase>> dc:source ?src
}
```

When used in this manner, there is an implicit variable binding for the embedded statement. Note that this query may be answered efficiently. For example, one query plan is:

- 2-bound POS index scan (?who :bought sybase) => ?sid
- JOIN (?sid dc:source ?src)

The following is an alternative syntax makes that variable binding explicit and allows for its reuse:

```
SELECT ?src ?who ?created {
   <<?who :bought :sybase>> as ?sid .
   ?sid dc:source ?src
   OPTIONAL {?sid dc:created ?created}
}
```

[41] We have spoken with a number of RDF vendors and RDF customers. They are universally in favor of this simplification. You can always use the RDF Reification syntax if you do not want this implication.

[42] The indexing of the statements about statements is a *database schema choice*. It should be completely transparent to database users. In fact, a database can transparently translate both reified statements and reified statement patterns into an internal format that is more efficient for indexing and query without offering the more pleasant syntax. The syntax is a sugar coating that makes it much more pleasant to deal with RDF Reification by eliminating some very ugly syntax structures.

The binding between the triple pattern and the ?sid variable can work in either direction. Given a binding for ?sid, it can decompose it into the (s,p,o) components of the bound statement. Given a statement expressed as (s,p,o) components, it can compose the inline representation of that statement and bind it on the variable ?sid. This allows easy bi-directional composition and decomposition of statements in a manner that is compatible with quads.

Conclusion

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 [43], techniques to counteract latency in distributed query (symmetric hash joins [44] and eddies [45]) and query against open web [46] [47], including frameworks with the potential to support critical thinking about data on the open web [48], including reasoning about evidence supporting conflicting conclusions, unreliable conclusions, and conclusions relying on incomplete evidence [49] [50] [51] [52]. Another line of research on *schema agnostic query* [53] explores *how* people can ask questions

---

[43] 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.

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

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

[46] 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.

[47] 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.

[48] 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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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* [54]. *Graph mining* is concerned with discovering, identifying, aggregating, and summarizing interesting patterns in graphs. As in schema agnostic query, people tying 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 [55] [56] [57] [58] using multiple full traversals over the graph, and decomposed over parallel hardware. SYSTAP, LLC is currently leading a team of researchers to develop a capability for *graph search and graph mining* on GPGPUs. GPGPUs are massively parallel hardware originally developed to accelerate video processing for games, and now used in cell phones and the world's largest super computer [59]. This will be an open source project under a liberal license. We plan to integrate this work into the bigdata platform, providing a seamless capability for linked data, structured graph query, graph search, and graph mining. We also see this as an opportunity to apply GPUs to computational models of cognition [60] [61] [62] in support of large-scale open collaboration frameworks and mapping the human connectome [63] [64] [65].

---

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

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

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

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

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

[59] The Titan Supercomputer installation at the Oak Ridge National Laboratory achieves 20 Petaflops using 299,009 Opteron cores and 18,688 GPUs. 16 Opteron cores per node. 1 GPU per node. Each GPU has 2,496 CUDA cores delivering 3.52 Teraflops per GPU.

[60] Shastri, L and Mani, D.R. (1997) "Massively parallel knowledge representation and reasoning: Taking a cue from the brain". In Parallel Processing for Artificial Intelligence 3 (Eds) Geller, J., Kitano, H. and Suttner, C. Elsevier Science.

[61] Shastri, L. Recent Advances in Shruti. In, (Eds.) Fredric Maire, Ross Hayward, A Connectionist Systems For Knowledge Representation and Deduction. Joachim Diederich. Queensland University of Technology, Neurocomputing Research Center. 1997. (pp. 1-14).

[62] Thompson, B.B., Cohen, M. S., and Freeman, J.T. (1995). Metacognitive Behavior in Adaptive Agents. *Proceedings of the World Congress on Neural Networks*.

[63] Sporns O, Tononi G, Kotter R: The Human Connectome: A Structural Description of the Human Brain. PLoS Comput Biol 2005, 1(4):e42.

[64] Vogelstein, Joshua T. "Q&A: What is the Open Connectome Project?." *Neural Systems & Circuits* 1.1 (2011): 1-4.

[65] Wikipedia contributors. "Brain Activity Map Project." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 28 Feb. 2013. Web. 28 Feb. 2013.