Presentation outline

- Bigdata® and the Semantic Web
- Bigdata® Architecture
  - Indices and Dynamic Sharding
  - Services, Discovery, and Dynamics
  - Bulk Data Loader
- Bigdata® RDF Database
  - Provenance mode
  - Vectored Query Engine
  - Analytic Query mode
  - Performance
  - Roadmap
- Sample applications
We focus on the backend semantic web database architecture and offer support and other services around that.

Reasons for working with SYSTAP, include:

- You have a prototype and you want to get it to the market.
- You want to embed a fast high performance semantic web database into your application.
- You want to integrate, query, and analyze large amounts information across the enterprise.
High level take aways

- Fast, scalable, open source, standards compliant database
  - Single machine to 50B triples or quads
  - Scales horizontally on a cluster
  - Compatible with 3rd party inference and middle tier solutions
- Reasons to be interested:
  - You need to embed a high performance semantic web database.
  - You need to integrate, query, and analyze large amounts information.
  - You need fast turn around on support, new features, etc.
What is “big data?”

• Big data is a way of thinking about and processing massive data.
  – Petabyte scale
  – Distributed processing
  – Commodity hardware
  – Open source

And with the emergence of big data analytics, many core and massively parallel processing.
Different kinds of “big” systems

- Parallel File Systems and Blob Stores
  - GFS, S3, HDFS, etc.
- Map / reduce
  - Good for data locality in inputs
    - E.g., documents in, hash-partitioned full text index out.
- Row stores
  - High read / write concurrency using atomic row operations
  - Basic data model is
    - { primary key, column name, timestamp } : { value }
  - Useful for primary key access to large data sets
- Parallel (clustered) databases
  - The Bigdata® platform fits into this category.

There are many different approaches to massive data. I have listed a few here.

**Row stores**

Row stores provide single key access to schema flexible data. The original inspiration was Google’s “bigtable” system.

**Map reduce**

Map reduce decompose a problem, performs a local computation, and then aggregates the outputs of the local computation. Map reduce is great for problems with good input locality. The general map/aggregation pattern is an old one. The modern map/reduce systems are inspired by Google’s “map/reduce” system.

**Main memory graph processing**

These systems provide in-memory graph processing, but lack many of the features of a database architecture (isolation, durability, etc). The intelligence community has a long history in this area. More recently “RAM clouds” are emerging in the private sector. There are also custom hardware solutions in the space, such as the Cray XMT.

**Parallel databases**

Parallel database platforms offer ACID (Atomic, Consistent, Isolated, Durable) data processing. This world can be divided up again into traditional relational databases, column stores, and increasingly, semantic web database platforms.
The Semantic Web

- The Semantic Web is a stack of standards developed by the W3C for the interchange and query of metadata and graph structured data.
  - Open data
  - Linked data
  - Multiple sources of authority
  - Self-describing data and rules
  - Federation or aggregation
  - And, increasingly, provenance

As a historical note, *provenance* was left out of the original semantic web architecture because it requires statements about statements which implies second order predicate calculus. At the Amsterdam 2000 W3C Conference, TBL stated that he was deliberately staying away from second order predicate calculus. Provenance mechanisms are slowly emerging for the semantic web due to their absolute necessity in many domains. The SGML / XML Topic Maps standardization was occurring more or less at the same time. It had a focus which provided for provenance and semantic alignment with very little support for reasoning. Hopefully these things will soon be reconciled as different “profiles” reflecting different application requirements.

Provenance is huge concern for many communities. However, support for provenance, other than digitally signed proofs, was explicitly off the table when the semantic web was introduced in 1999. TBL's reason for not tackling provenance was that it requires second order predicate calculus, while the semantic web is based on first order predicate calculus. However, it is possible to tackle provenance without using a highly expressive logic. The ISO and XML Topic Maps communities have lead in this area and, recently, there has been increasing discussion about provenance within the W3C.
English: The […] diagram [above] visualizes the data sets in the LOD cloud as well as their interlinkage relationships. Each node in this cloud diagram represents a distinct data set published as Linked Data. The arcs indicate that RDF links exist between items in the two connected data sets. Heavier arcs roughly correspond to a greater number of links between two data sets, while bidirectional arcs indicate the outward links to the other exist in each data set.

Image by Anja Jentzsch. The image file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.
This is one system which is using bigdata. There are more slides on KaBOB at the end of the deck. There are currently 8.7B statements drawn from 17 databases and stored in a single bigdata Journal. KaBOB differs from many efforts in its intent to create a high quality curated database over which high level reasoners may be run. In order to run high level reasoners, the data needs to be very carefully reviewed with respect to its semantics. For example, making sure that the distinction between a gene, a gene expression, and an experiment which indicates some correlation between a gene and a gene expression are all carefully distinguished. Without that, a high level reasoner will produce garbage rather than interesting entailments.
Enabling trends for bigdata®

- Open source movement
- Cloud infrastructure
- Price pressure on commodity hardware
- Many core computing (CPUs, GPUs)
- Solid state disk
- Semantic technologies and linked open data
- Business pressure toward increasing data scale

10 years ago everyone knew that the database was a frozen market. Today, nothing could be further from the truth. This is a time of profound upheaval in the database world.

There are a number of trends which are driving this. One is the continued pressure on commodity hardware prices and performance. Another is the growth in open source software. Today, all “big data” systems leverage open source software and many federal contracts require it. At the same time the speed of processors has hit the wall so applications are forced to parallelize and use distributed architectures. SSD has also changed the playing field, virtually eliminating disk latency in critical systems.

The central focus for the bigdata® platform central focus is a parallel semantic web database architecture. There are other distributed platforms, including those based on custom hardware solutions, for main memory graph processing. These tend to lack core features of a database architecture such as durability and isolation.
The killer “big data” app

- Clouds + “Open” Data = Big Data Integration
- Critical advantages
  - Fast integration cycle
  - Open standards
  - Integrate heterogeneous data, linked data, structured data, and data at rest.
  - Opportunistic exploitation of data, including data which can not be integrated quickly enough today to derive its business value.
  - Maintain fine-grained provenance of federated data.

This is our take on where this is all heading. We tend to focus on high data scale and low expressivity with rapid data exploitation cycles.
bigdata®

- Petabyte scale
- Dynamic sharding
- Commodity hardware
- Open source, Java

• High performance
• High concurrency (MVCC)
• High availability
• Temporal database

Semantic web database
Key Differentiators

- **Scalable**
  - Single machine scaling to 50B triples/quads
  - Cluster scales-out incrementally using dynamic sharding

- **Temporal database**
  - Fast access to historical database states.
  - Light weight versioning.
  - Point in time recovery.

- **Highly available**
  - Built in design for high availability.

- **Open source**
  - Active development team including OEMs and Fortune 500 companies.
Petabyte scale

- Metadata service (MDS) used to locate shards.
- Maps a key range for an index onto a shard and the logical data service hosting that shard.
- MDS is heavily cached and replicated for failover.
- Petabyte scale (tera-triples) is easily achieved.
- Exabyte scale is much harder; breaks the machine barrier for MDS.

<table>
<thead>
<tr>
<th>MDS scale</th>
<th>Data scale</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>71MB</td>
<td>terabyte</td>
<td>18,000,000,000</td>
</tr>
<tr>
<td>72GB</td>
<td>petabyte</td>
<td>18,000,000,000,000</td>
</tr>
<tr>
<td>74TB</td>
<td>exabyte</td>
<td>18,000,000,000,000,000</td>
</tr>
</tbody>
</table>
Large scale systems need to think about deployment architectures and managing the raw resources that they will consume. Bigdata® has evolved over time into a hybrid architecture in which there is a separation of concerns between the compute and persistence layers. The architecture still leverages local computation (including local disk access) whenever possible. We do this now through caching shards on the instances nodes while leaving read-only files on long term durable storage. This architecture makes it possible to deploy to platforms like Amazon EC2 (compute) + Amazon S3 (high 9s durable storage).
Bigdata®

Services and dynamics
bigdata® is a federation of services

Data services (DS) manage shards
Shard locator service (SLS) maps shards to their data services
Transaction service (TS) coordinates transactions
Load balancer service (LBS) centralizes load balancing decisions
Client services (CS) provide execution of distributed jobs
Jini lookup services (LUS) provide service discovery
Zookeeper quorum servers (ZK) provide for master election, configuration management, and global synchronous locks
Typical Software Stack

- Application Layer
  - Unified API
  - Unified API Implementation
  - API Frameworks (Spring, etc.)
- Java
  - Sesame Framework
    - SAIL
    - RDF
    - SPARQL
  - Bigdata RDF Database
  - Bigdata Component Services
- OS (Linux)
- Cluster and Storage Management
Service Discovery (1/2)

- Services discover registrars. Registrar discovery is configurable using either unicast or multicast protocols.
- Services advertise themselves and lookup other services (a).
- Clients use the shard locator to locate key-range shards for scale-out indices (b).

Client Service → Registrar (a)

Registrar → Shard Locator (b)

Shard Locator → Data Service

Data Service

Data Service

Data Service

RMI / NIO Bus
Service Discovery (2/2)

- Clients resolve shard locators to data service identifiers (a), then lookup the data services in service registrar (b).
- Data moves directly between clients and services (c).
- Service protocols not limited to RMI. Custom NIO protocols for data high throughput.
- Client libraries encapsulate this for applications, including caching of service lookup and shard resolution.
Persistence Store Mechanisms

- **Write Once, Read Many (WORM) store**
  - Append-only, log structured store (aka “journal”)
  - Used by the data services used to absorb writes
  - Plays an important role in the temporal database architecture

- **Read/Write (RW) store**
  - Efficiently recycles allocation slots on the backing file
  - Used by services that need persistence without sharding
  - Also used in the scale-up single machine database (~50B triples)

- **Index segment (SEG) store**
  - Read-optimized B+Tree files
  - Generated by bulk index build operations on a data service
  - Plays an important role in dynamic sharding and analytic query

There are three distinct persistence store mechanisms within bigdata:
- Write Once, Read Many (WORM) store, which is based an append-only, logstructured store and is by the data services used to absorb writes. This is often referred to as the “journal”;
- Read/Write (RW) store, which is capable of reusing allocation slots on the backing file and is used for services where we want to avoid decomposing the data into managed shards; and
- Index segment (SEG) store, which is a read-optimized B+Tree file generated by a bulk index build operation on a data service.

Both the WORM and RW persistence store designs support write replication, record level checksums, low-level correction of bad reads, and resynchronization after service joins a failover chain. The index segments are only present on the data service nodes and are part of the mechanisms for dynamic sharding. Once an index segment file has been generated, it is replicated to the other physical data service nodes for the same logical data service instance.

Bigdata provides a strong guarantee of consistency for the nodes in a failover chain such that any physical service instance in the chain may service any request for any historical commit point which has not be released by the transaction manager. This allows us to use shard affinity among the failover set for a given logical data service to load balance read operations. Since the members of the failover group have consistent state at each commit point, it possible for any node in the failover set to perform index segment builds. In general, it makes sense to perform the build on the node with the strongest affinity for that shard as the shard’s data is more likely to be cached in RAM.
RWStore

- Read-write database architecture
  - Scales to ~ 50B triples.
  - Fast
- Manages a pool of allocation regions
  - 1k, 2k, 4k, 8k, etc.
  - Reasonable locality on disk through management of allocation pools, but
    - Requires SCSI (or SSD) for fast write performance (SATA is not good enough).
- Tracks deleted records (B+Tree nodes and leaves)
  - Releases them once the commit points from which they were visible have
    been expired.
- Commit points expire when:
  - No transactions open against any commit point LTE ts.
  - (now – ts) GTE retention time
Once clients have discovered the metadata service, they can begin operations against data services. This diagram shows how writes originating with different clients can get applied to a data service, how the data services use a write through pipeline to provide data replication, and how the writes appear on the append only journal and then migrate over time onto a read-optimized index segment.

The concepts of the append only, fully-buffered journal and overflow onto read-optimized index segments are basic building blocks of the architecture.
If a bigdata federation is configured to retain large amounts of (or infinite) history, then the persistent state of the services must be stored on a parallel file system, SAN, or NAS.

This is a shared disk architecture. While quorums are still used in the shared disk architecture configuration, the persistent state is centralized on inherently robust managed storage. Unlike the shared nothing architecture, synchronization of a node is very fast as only the live journal must be replicated to a new node (see below).

In order to rival the write performance of the shared nothing architecture, the shared disk architecture uses write replication for the live journal on the data services and stores the live journal on local disk on those nodes. Each time the live journal overflows, the data service opens a new live journal, and closes out the old journal against further writes. Asynchronous overflow processing then replicates the old journal to shared disk and batch builds read-only B+Tree files (index segments) from the last commit point on the
Bigdata® Indices

Scale-out B+Tree and Dynamic Sharding
Bigdata® Indices

- Dynamically key-range partitioned B+Trees for indices
  - Index entries (tuples) map unsigned byte[ ] keys to byte[ ] values.
  - “deleted” flag and timestamp used for MVCC and dynamic sharding.
- Index partitions distributed across data services on a cluster
  - Located by centralized metadata service

<table>
<thead>
<tr>
<th>nodes</th>
<th>c</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>sep. keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>child refs</td>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>leaves</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>keys</td>
<td>1 2 3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>values</td>
<td>v1 v2 v3</td>
<td>v4 v5 v6</td>
<td>v7 v8 v9 v10</td>
</tr>
</tbody>
</table>

Tuples
- key : unsigned byte (*)
- val : byte (*)
- version : long
- deleted : boolean
Dynamic Key Range Partitioning

Splits break down the shards as the data scale increases.

Joins merge shards when data scale decreases.

Moves redistribute shards onto existing or new nodes in the cluster.
Dynamic Key Range Partitioning

Initial conditions place the first shard on an arbitrary data service representing the entire index key range.

Shard Locator Service

 DataService1

\( (\emptyset, \infty) \)
Dynamic Key Range Partitioning

Writes cause the shard to grow. Eventually its size on disk exceeds a preconfigured threshold.

Shard Locator Service

 DataService1

p0

$([], \infty)$
Instead of a simple two-way split, the initial shard is "scatter-split" so that all data services can start managing data.

Dynamic Key Range Partitioning

Shard Locator Service

 DataService1

\([0, \infty)\)
The newly created shards are then dispersed around the cluster.

Subsequent splits are two-way and moves occur based on relative server load (decided by Load Balancer Service).
Shard Evolution

**Builds** generate index segments from just the old journal.

**Merge** compacts the shard view into a single index segment.

- journal\_0
- build -> p0
- journal\_1
- p0
- build -> p1
- journal\_2
- p1
- build -> p2
- journal\_3
- p2
- p1
- p0
- merge -> p3
Shard Evolution

- Initial journal on DS.
- Incremental build of new segments for a shard with each journal overflow.
- Shard periodically undergoes compacting merge.
- Shard will split at 200MB.
Bulk Data Load

High Throughput with Dynamic Sharding
Bulk Data Load

- Very high data load rates
  - 1B triples in under an hour (better than 300,000 triples per second on a 16 node cluster).
- Executed as a distributed job
  - Read data from a file system, the web, HDFS, etc.
- Database remains available for query during load
  - Read from historical commit points with snapshot isolation.
Batch Online or Offline

- The current batch load is online.
  - Snapshot isolation is available for consistent read behind
  - Readers advance to new global commit point after load
- Batch offline could also be achieved
  - 1st Map/Reduce pass
    - Assign identifiers and compute key-range partitions
  - 2nd Map/Reduce pass builds
    - Covering indices (3 or 6)
    - DESCRIBE cache (a key-value cache for RDF subjects)
  - Global cutover to new federation instance
  - Offline bulk load would provide all the same query features and much higher load rates.
This is an “incremental” bulk load into an existing system with snapshot isolation for consistent read behind against the same cluster.
The bottleneck was the NSF mount from which the data were read.
The bottleneck was the NFS mount from which the data were read.
Disk utilization under sustained write activity converges to a steady state after ~ 2B triples. These numbers are without block compression in the index segments.

The initial peak and the peak and valley pattern are related to full GCs in the JVM.
Shards over time (2.5B triples)

Total Shard Count on Cluster over time for 2.5B triples

- Shards vs. minutes graph showing the increase in shard count over time.
This graph shows the relative frequency of shard build, merge, and split events. Shard **builds** simply transfer the last commit point of an index onto a new index segment file. This action puts the data in key order on a read-only file. By building the index segment files, we can also release the journal on which the original writes were buffered (once the retention period for commit points on that journal has expired). Shard **merges** compact the total view of a shard into a single index segment file. A shard merge is executed used a priority queue based on the complexity of the shard view. Shard **splits** break an index file into one or more key ranges. A shard split is executed once the size on disk of a compact shard exceeds a threshold (200MB).
This is an interesting view into the early life cycle of shards on a cluster. At the start of the timeline, each index (SPO, OSP, POS, T2ID, ID2T) is on one node. These initial splits are accelerated by a low threshold for the size of a “full” shard on the disk. As a result, those initial index partitions are split very quickly. The initial shard split is a “scatter split”. The initial shard of each index is split into N*2 shards, where N is the #of nodes in the cluster. Those shards are then scattered across the cluster. The top half of the graph shows the shard split events. The bottom half of the graph shows the shard move events, which is how the data are redistributed. Throughput accelerates rapidly as the system goes through these initial scatter splits.
Bigdata® RDF Database
Bigdata 1.2

- Scales to 50B triples/quads on a single machine.
  - Scales horizontally on a cluster.
  - Easy to embed in 3rd party applications.

- SPARQL 1.1 support (except property paths)
  - Query, Update, Federated Query, Service Description.
  - Native RDFS+ inference.
  - Analytic query.

- Three database modes:
  - triples, provenance, or quads.

- Bigdata 1.1 largely focused on query performance.
- Put all of our effort into something that really scales up.
Recent Features

- Analytics package
  - Native memory manager scales up to 4TB of RAM and eliminates GC overhead. Still 100% Java.
  - New hash index data structure for named solution sets and hash joins.
- New query model
  - Faster query plans.
  - “Explain” a query.
- Embedded SERVICEs.
  - Query extensions, custom indexing, monitor updates, etc.
- Query hints using virtual triples
  - Fine grained control over evaluation order, choice of operators, query plans, etc.
- Storage layer improvements:
  - New index for BLOBs.
  - Inline common vocabulary items in 2-3 bytes.
RDF Database Modes

- **Triples**
  - 2 lexicon indices, 3 statement indices.
  - RDFS+ inference.
  - All you need for lots of applications.

- **Provenance**
  - Datum level provenance.
  - Query for statement metadata using SPARQL.
  - No complex reification.
  - No new indices.
  - RDFS+ inference.

- **Quads**
  - Named graph support.
  - Useful for lots of things, including some provenance schemes.
  - 6 statement indices, so nearly twice the footprint on the disk.
RDF Database “Schema” (1/2)

- Dynamically key-range sharded indices:
- Lexicon relation maps RDF Values (terms) into compact internal identifiers.
- Statement relation provides fast indices for SPARQL query.
- Scale-out indices gives us scale-out RDF

bigdata® RDF store

Lexicon Relation
- t2id
- id2t
- blobs

Statement Relation
- spo
- pos
- osp

bigdata® RDF

bigdata® RDF store

“Schema” (1/2)
RDF Database “Schema” (2/2)

This is slightly dated. We now inline numeric data types into the statement indices for faster aggregation and use a BLOBS index for large literals.

The tables and provenance indices use three statement indices.

The quad store uses six indices whose keys are based on permutations of (p, o, c, v).
Covering Indices (Quads)

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Sky</td>
<td>is colored</td>
<td>“Blue”</td>
<td>(Context)</td>
</tr>
</tbody>
</table>

- Context, Subject, Predicate, Object
  - [Context] The Sky is colored “Blue” (Context)
- Subject, Predicate, Object, Context
  - The Sky is colored “Blue” (Context)
- Object, Context, Subject, Predicate
  - “Blue” (Context) The Sky is colored
- Predicate, Object, Context, Subject
  - is colored “Blue” (Context) The Sky
- Predicate, Context, Subject, Object
  - is colored “Blue” (Context) The Sky

---

bigdata®
Presented at Graph Data Management 2012
© 2006-2012 All Rights Reserved
http://www.bigdata.com/blog
Statement Level Provenance

- Important to know where data came from in a mashup

- `<mike, memberOf, SYSTAP>`
- `<http://www.systap.com, sourceOf, ....>`

- But you **CAN NOT** say that in RDF.
RDF “Reification”

• Creates a “model” of the statement.

\[
_{-s1, \text{subject, :mike}} \\
_{-s1, \text{predicate, :memberOf}} \\
_{-s1, \text{object, :SYSTAP}} \\
_{-s1, \text{type, Statement}}
\]

• Then you can say,

\[
\langle \text{http://www.systap.com, :sourceOf, }_{-s1} \rangle
\]
bigdata® Statement Identifiers (SIDs)

- Statement identifiers let you do exactly what you want:
  
  `<:mike, :memberOf, :SYSTAP, _s1>`
  `
  `<<http://www.systap.com>, :sourceOf, _s1>>`

- SIDs look just like blank nodes

- And you can use them in SPARQL

  construct { ?s :memberOf ?o . ?s1 ?p1 ?sid . }
  where {
    GRAPH ?sid { ?s :memberOf ?o }
  }
Quads with Provenance

- Recasting the same idea to support quads with provenance.
  - This would be a new feature. We do not currently support statement level provenance in the quads mode.
- Bigdata represents SIDs by *inlining* statements into statements.
  - \(<<s,p,o>,p1,o1>\)
- SPARQL extension functions support SIDs in any mode:
  - boolean:isSid(varOrConst)
    - true iff the argument evaluates to a SID.
  - sid(sid,s,p,o,[c])
    - where args are variables or constants.
    - Unifies sid with \(<s,p,o,[c]>\). Used to (de-)compose statements about statements.

See https://sourceforge.net/apps/trac/bigdata/ticket/526
bigdata® Statement Identifiers (SIDs)

- Using sid() in SPARQL.

```
construct { ?s :memberOf ?o . ?s1 ?p1 ?sid . }
where {
  sid(?sid,?s,memberOf,?o,?g) .
  GRAPH ?g { ?s :memberOf ?o }
}
```

» SPARQL functions might need to look a little different since SPARQL does not support unification (it can not bind function arguments).
Bigdata® Vectored Query Engine
Bigdata Query Engine

- Vectored query engine
  - High concurrency and vectored operator evaluation.

- Physical query plan (BOPs)
  - Supports pipelined, blocked, and at-once operators.
  - Chunks queue up for each operator.
  - Chunks transparently mapped across a cluster.

- Query engine instance runs on:
  - Each data service; plus
  - Each node providing a SPARQL endpoint.
Query Plan Improvements

- **Analytic Query Mode**
  - Large intermediate results with ZERO GC.
- **Sub-Select, Sub-Groups, and OPTIONAL groups**
  - Based on a hash join pattern, can be 100s of times faster.
  - Automatically eliminates intermediate join variables which are not projected.
- **Merge join pattern**
  - Used when a series of group graph patterns share a common join variable.
- **Pipelined aggregation**
  - Does not materialize data unless necessary
- **Named solution sets**
  - An Anzo extension.
- **Faster default graph queries**
- **Query hints**
  - Control join order and many other things.
Analytic Package

- Bigdata® Memory Manager
  - 100% Native Java
  - Application of the bigdata® RWStore technology
  - Manages up to 4TB of the native process heap — No GC.
  - Relieves heap pressure, freeing JVM performance.

- Extensible Hash Index (HTree)
  - Fast, scalable hash index
  - Handles key skew and bucket overflow gracefully
  - Use with disk or the memory manager (zero GC cost up to 4TB!)
  - Used for analytic join operators, named solution sets, caches, etc.

- Runtime Query Optimizer — coming soon.

Go and get sesame, it will fall over because of this.

Several commercial and open source Java grid cache products exist, including Oracle’s Coherence, infinispan, and Hazelcast. However, all of these products share a common problem – they are unable to exploit large heaps due to an interaction between the Java Garbage Collector (GC) and the object creation and object retention rate of the cache. There is a non-linear interaction between the object creation rate, the object retention rate, and the GC running time and cycle time. For many applications, garbage collection is very efficient and Java can run as fast as C++. However, as the Java heap begins to fill, the garbage collector must run more and more frequently, and the application is locked out while it runs. This leads to long application level latencies that bring the cache to its knees and throttles throughput. Sun and Oracle have developed a variety of garbage collector strategies, but none of them are able to manage large heaps efficiently. A new technology is necessary in order to successfully deploy object caches that can take advantage of servers with large main memories.
Choose standard or analytic operators

- Easy to specify which
  - URL query parameter or SPARQL query hint

- Java operators
  - Use the managed Java heap.
  - Can sometimes be faster or offer better concurrency
    - E.g., distinct solutions is based on a concurrent hash map
  - BUT
    - The Java heap can not handle very large materialized data sets.
    - GC overhead can steal your computer

- Analytic operators
  - Scale up gracefully
  - Zero GC overhead.
Query Hints

- Virtual triples
  - Scope to query, graph pattern group, or join.
- Allow fine grained control over
  - Analytic query mode
  - Evaluation order
  - Vector size
  - Etc.

```sparql
SELECT ?x ?y ?z
WHERE {
  hint:Query hint:vectorSize 10000 .
  ?x a ub:GraduateStudent .
  ?y a ub:University .
  ?z a ub:Department .
  ?x ub:memberOf ?z .
  ?x ub:undergraduateDegreeFrom ?y .
}
```
New Bigdata Join Operators

- Pipeline joins (fast, incremental).
  - Map binding sets over the shards, executing joins close to the data.
  - Faster for single machine and much faster for distributed query.
  - First results appear with very low latency
- Hash joins against an access path
  - Scan the access path, probing the hash table for joins.
  - Hash joins on a cluster can saturate the disk transfer rate!
- Solution set hash joins
  - Combine sets of solutions on shared join variables.
  - Used for Sub-Select and Group Graph Patterns (aka join groups)
- Merge joins
  - Used when a series of group graph patterns share a common join variable
    - Think OPTIONALs.
  - Java: Must sort the solution sets first, then a single pass to do the join.
  - HTree: Linear in the data (it does not need to sort the solutions).
As part of our 1.1 release, we had to take over more control of query evaluation from the Sesame platform. We found that that Sesame operator tree was discarding too much information about the structure of the original SPARQL query. Also, bigdata query evaluation has always been vectored and operators on IVs rather than RDF Values. The Sesame visitor pattern was basically incompatible with bigdata query evaluation. Bigdata is now integrated with the Sesame platform directly at the SPARQL parser. All query evaluation from that point forward is handled by bigdata.

Most of the work is now done using the Abstract Syntax Tree (AST). The AST is much closer to the SPARQL grammar. AST optimizers annotate and rewrite the AST. AST then translated to Bigdata Operators (BOPs). BOPs run on the query engine. The AST ServiceNode provides for 3rd party operators and services.

The bigdata query engine is vectored and highly concurrent. Most bigdata operators support concurrent evaluation and multiple instances of the same operator can be running in parallel. This allows us to fully load the disk queue. With the new memory manager and analytic query operators, having large solutions sets materialized in memory no longer puts a burden on the JVM.
Data Flow Evaluation

- Left-to-right evaluation
  - Physical query plan is operator sequence (not tree)

![Diagram: op1 → op2 → op3]

- Parallelism
  - Within operator
  - Across operators
  - Across queries

- Operator annotations used extensively.
“Pipeline” Joins

- Fast, vectored, indexed join.
- Very low latency to first solutions.

- Mapped across shards on a cluster.
  - Joins execute close to the data.
- Faster for single machine
  - Much faster for distributed query.
Preparing a query

Original query:

```
SELECT ?x WHERE {
  ?x a ub:GraduateStudent ;
  ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0> .
}
```

Translated query:

```
query := (x 8 256) ^ (x 400 3048)
```

Query execution plan (access paths selected, joins reordered):

```
query := pos(x 400 3048) ^ spo(x 8 256)
```
Pipeline Join Execution
• Solutions flow to both the default and alt sinks.
• UNION(A,B)

TEE

• Generalizes to n-ary UNION
Conditional Flow

- Data flows to either the default sink or the alt sink, depending on the condition.

- Branching pattern depends on the query logic.
The JVM and HTree hash join utility classes understand the following “kinds” of hash join:

/**
* A normal join. The output is the combination of the left and right hand
* solutions. Only solutions which join are output.
* /
* Normal,
**/
* An optional join. The output is the combination of the left and right
* hand solutions. Solutions which join are output, plus any left solutions
* which did not join. Constraints on the join ARE NOT applied to the
* "optional" path.
* /
* Optional,
**/
* A join where the left solution is output iff there exists at least one
* right solution which joins with that left solution. For each left
* solution, that solution is output either zero or one times. In order to
* enforce this cardinality constraint, the hash join logic winds up
* populating the "join set" and then outputs the join set once all
* solutions which join have been identified.
* /
* Exists,
**/
* A join where the left solution is output iff there is no right solution
* which joins with that left solution. This basically an optional join
* where the solutions which join are not output.
* <p>
* Note: This is also used for "MINUS" since the only difference between
* "NotExists" and "MINUS" deals with the scope of the variables.
* /
* NotExists,
**/
* A distinct filter (not a join). Only the distinct left solutions are
* output. Various annotations pertaining to JOIN processing are ignored
* when the hash index is used as a DISTINCT filter.
*/
Named Solution Sets

- Support solution set reuse

- Evaluation
  - Run before main WHERE clause.
  - Exogenous solutions fed into named subquery.
  - Results written on hash index.
  - Hash index INCLUDED into query.

- Example is BSBM BI Q5

```
WITH {
  SELECT ?country ?product (count(?review) AS ?nrOfReviews)
  WHERE {
    ?review bsbm:reviewFor ?product .
  }
  GROUP BY ?country ?product
} AS %namedSet1

WHERE {
  { SELECT ?country (max(?nrOfReviews) AS ?maxReviews)
    INCLUDE %namedSet1
  } GROUP BY ?country
  { SELECT ?product (avg(xsd:float(?price)) AS ?avgPrice)
    INCLUDE %namedSet1
  } GROUP BY ?product
} INCLUDE %namedSet1 .
FILTER(?nrOfReviews = ?maxReviews)
ORDER BY desc(?nrOfReviews) ?country ?product
```

See
Bottom Up Evaluation

- SPARQL semantics specify “as if” bottom up evaluation
  - For many queries, left-to-right evaluation produces the same results.
  - Badly designed left joins must be lifted out and run first.

```
SELECT * {
  :x1 :p ?v .
  OPTIONAL {
    :x3 :q ?w .
    OPTIONAL { :x2 :p ?v }
  }
}
```

```
SELECT * {
  WITH {
    SELECT ?w ?v {
      :x3 :q ?w .
      OPTIONAL { :x2 :p ?v }
    }
  } AS %namedSet1
  :x1 :p ?v .
  OPTIONAL {
    INCLUDE %namedSet1
  }
}
```
Merge Join Pattern

- High performance n-way join
  - Linear in the data (on HTree).
- Series of group graph patterns must share a common join variable.
- Common pattern in SPARQL
  - Select something and then fill in optional information about it through optional join groups.

```
SELECT (SAMPLE(?var9) AS ?var1) ?var2 ?var3
{
  ?var3 rdf:type pol:Politician.
  ?var3 pol:hasRole ?var6.
  ?var6 pol:party "Democrat".
}
OPTIONAL {
  ?var3 p1:name ?var9
}
OPTIONAL {
}
...
GROUP BY ?var2 ?var3
```

Really common pattern in SPARQL – select something and then fill in optional information about it through optional join groups.

The critical pattern for a merge join is that we have a hash index against which we may join several other hash indices. To join, of course, the hash indices must all have the same join variables. The pattern recognized here is based on an initial INCLUDE (defining the first hash index) followed by several either required or OPTIONAL INCLUDEs, which are the additional hash indices. The merge join itself is the N-way solution set hash join and replaces the sequence of 2-way solution set hash joins which we would otherwise do for those solution sets. It is more efficient because the output of each 2-way join is fed into the next 2-way join, which blows up the input cardinality for the next 2-way hash join. While the merge join will consider the same combinations of solutions, it does so with a very efficient linear pass over the hash indices. (For the JVM Merge Join operator we have to do a SORT first, but the HTree imposes a consistent ordering by the hash bits so we can go directly to the linear pass. (It is actually log linear due to the tree structure of the HTree, but it is against main memory and it is a sequential scan of the index so it should be effectively linear.)

This is based on query21 from our govtrack CI dataset.
Merge Joins (cont.)

```
SELECT (SAMPLE(?var9) AS ?var1) ?var2 ?var3
{
    SELECT DISTINCT ?var3
    WHERE {
        ?var3 rdf:type pol:Politician.
        ?var3 pol:hasRole ?var6.
        ?var6 pol:party "Democrat".
    }
}
OPTIONAL {
    ?var3 p1:name ?var9
}
OPTIONAL {
}
...
GROUP BY ?var2 ?var3
```

Two implementations:

Java : Sorts the solution sets first, then does a linear scan for the join.

HTree : Linear in the data (htree imposes a consistent ordering).
Parallel Graph Query Plans

- Partly ordered scans
  - Parallel AP reads against multiple shards
- Hash partitioned
  - Distinct
  - Order by
  - Aggregation (when not pipelined)
- Default Graph Queries
  - Global index view provides RDF “Merge” semantics.
  - DISTINCT TRIPLES across the set of contexts in the “default graph”

We do only some of this today, but this is where we are heading with the clustered database. Other than the “default graph” access path, this is all pretty standard stuff.

We are looking at alternative ways to handle default graph queries on a cluster.
Parallel Hash Joins for Unselective APs

- AP must read a lot of data with reasonable locality
  - Shard view: journal + index segment(s).
    - One IO per leaf on segments.
    - 90%+ data in key-order on disk
    - Narrow stride (e.g., OSP or POS)
- Crossover once sequential IO dominates vectored nested index join
  - Key range scans turn into sequential IO in the cluster
  - Goal is to saturate the disk read bandwidth
- Strategy A
  - Map intermediate solutions against target shards
  - Build hash index on target nodes
  - Parallel local AP scans, probing hash index for joins.
- Strategy B
  - Hash partition intermediate solutions on join variables.
  - Parallel local AP scans against spanned shard(s)
    - Vectored hash partition of solutions read from APs.
    - Vectored probe in distributed hash index partitions for joins.

The question is how to best leverage the fact that 99%+ of the data are in key order on the disk in the clustered database architecture. We are looking at two basic designs here. One would map the intermediate solutions to the shard against which they need to be joined. The other would scan the shards, hash partitioning both the intermediate solutions and the solutions from the shard APs. Either way, the goal is to saturate the local disk when reading on unselective access paths in support of fast analytic query. This bears some relationship to column stores, not in the database architecture but in the goal of maximizing the IO transfer rate through narrow strides (triples or quads are quite narrow) and doing sequential IO when scanning a “column.”
Integration Points
Sesame Sail & Repository APIs

- Java APIs for managing and querying RDF data
- Extension methods for bigdata:
  - Non-blocking readers
  - RDFS+ truth maintenance
  - Full text search
  - Change log
  - Etc.
NanoSparqlServer

- Easy deployment
  - Embedded or Servlet Container
- High performance SPARQL end point
  - Optimized for bigdata MVCC semantics (queries are non-blocking)
  - Built in resource management
  - Scalable!
- Simple REST API
  - SPARQL 1.1 Query and Update
  - Simple and useful REST-ful INSERT/UPDATE/DELETE methods
  - ESTCARD exposes super fast range counts for triple patterns
  - “Explain” a query.
  - Monitor running queries.

Benefits over Sesame:
- trivial to deploy
- optimized for bigdata MVCC semantics
- built in resource management: controls the # of threads
SPARQL Query

- High level query language for graph data.
- Based on pattern matching and property paths.
- Advantages
  - Standardization
  - Database can optimize joins
    - Versus navigation-only APIs such as blueprints

SPARQL - http://www.w3.org/TR/sparql11-query/
Blueprints - https://github.com/tinkerpop/blueprints/wiki/
SPARQL Update

- **Graph Management**
  - Create, Add, Copy, Move, Clear, Drop

- **Graph Data Operations**
  - LOAD uri
  - INSERT DATA, DELETE DATA
  - DELETE/INSERT

  ```sparql
  ( WITH IRIref )?
  ( ( DeleteClause InsertClause? ) | InsertClause )
  ( USING ( NAMED )? IRIref )*
  WHERE GroupGraphPattern
  ```

- Can be used as a RULES language, update procedures, etc.
Bigdata SPARQL UPDATE

- Language extension for SPARQL UPDATE
  - Adds cached and durable solution sets.
    
    INSERT INTO %solutionSet1
    SELECT ?product ?reviewer WHERE { ... }

- Easy to slice result sets:
  
  SELECT ... { INCLUDE %solutionSet1 } OFFSET 0 LIMIT 1000
  SELECT ... { INCLUDE %solutionSet1 } OFFSET 1000 LIMIT 1000
  SELECT ... { INCLUDE %solutionSet1 } OFFSET 2000 LIMIT 1000

- Re-group or re-order results:
  
  SELECT ... { INCLUDE %solutionSet1 } GROUP BY ?x
  SELECT ... { INCLUDE %solutionSet1 } ORDER BY ASC(?x)

- Expensive JOINs are NOT recomputed.

See
Basic Federated Query

- Language extension for remote services
  
  SERVICE uri { graph-pattern }

- Integrated into the vectored query engine
  
  - Solutions vectored into, and out of, remote end points.
  - Control evaluation order query hints:
    
    - runFirst, runLast, runOnce, etc.

SERVICE { .... }
  hint:Prior hint:runFirst "true".

- ServiceRegistry

  - Configure service end point behavior

See


See https://sourceforge.net/apps/mediawiki/bigdata/index.php?title=QueryHints
Custom Services

- Examples
  - Bigdata search index
  - Ready made for 3rd party integrations (Lucene, spatial, embedded prolog, etc.)

- Custom Services
  - Run in the same JVM
  - Can monitor database updates
  - Invoked with the SERVICE graph pattern
  - Solutions vectored in and out.
Low Latency Full Text Index
Full Text Index

- Built on top of standard B+Tree index architecture
  - Enabled via a database property
  - Each literal added to database indexed by full text engine
- High level SPARQL integration
- Java API
  - Supports low latency paged result sets
- Supports whole word match and prefix match
  - Embedded wildcard search (other than prefix match) not supported
- Hits are scored by traditional cosine relevance
Full Text Index

- **Token** – extracted from RDF Literal.
  - Uses Lucene analyzers based on the language code.
  - Can register custom tokenizers, e.g., for engineering data
- **docId**
  - The IV of the RDF Literal having that token.
- **termFreq**
  - How many times the token appears in that literal
- **localTermWeight**
  - The normalized term frequency data.
Free Text Search: Basic

- Bigdata defines a number of “magic predicates” that can be used inside SPARQL queries:
- “bd:search” is used to bind a variable to terms in the full text index, which can then be used in joins:

```sparql
select ?s ?p o
where {
    # search for literals that use “mike”
    ?o bd:search "mike" .
    # join to find statements that use those literals
}
```
Free Text Search: Advanced

- Other magic predicates provide more advanced features:

```sparql
where {
  ?o bd:search "mike personick" .
  ?o bd:matchAllTerms "true" .
  ?o bd:minRelevance "0.25" .
  ?o bd:relevance ?score .
  ?o bd:maxRank "1000" .
  ?o bd:rank ?rank .
}
```
Benchmarks
LUBM U50

- Benchmark provides a mixture of select, unselective, simple, and complex queries.
- Can be run at many data scales.
- Queries are not parameterized, so all runs are “hot”.

<table>
<thead>
<tr>
<th>query</th>
<th>Time (ms)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>query1</td>
<td>38</td>
<td>4</td>
</tr>
<tr>
<td>query2</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>query3</td>
<td>42</td>
<td>25</td>
</tr>
<tr>
<td>query4</td>
<td>74</td>
<td>17</td>
</tr>
<tr>
<td>query5</td>
<td>77</td>
<td>1</td>
</tr>
<tr>
<td>query6</td>
<td>70</td>
<td>18</td>
</tr>
<tr>
<td>query7</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>query8</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>query9</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>query10</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>query11</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>query12</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>query13</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>query14</td>
<td>2.12</td>
<td>255,753</td>
</tr>
<tr>
<td>query15</td>
<td>2.246</td>
<td>350,115</td>
</tr>
<tr>
<td>query16</td>
<td>4,114</td>
<td>0</td>
</tr>
<tr>
<td>query17</td>
<td>4,114</td>
<td>0</td>
</tr>
<tr>
<td>query18</td>
<td>4,114</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>9,721</td>
<td></td>
</tr>
</tbody>
</table>
BSBM 100M

- Graph shows series of trials for the BSBM reduced query mix (w/o Q5).
- Metric is Query Mixes per Hour (QMpH). Higher is better.
- 8 client curve shows JVM and disk warm up effects. Both are hot for 16 client curve.
- Occasional low points are GC.
- Apple mini (4 cores, 16G RAM and SSD). Machine is CPU bound at 16 clients. No IO Wait.

Graph showing the comparison between 8 clients and 16 clients for QMpH metric.
Roadmap
### Bigdata® Roadmap

- **2012 Q1**
  - SPARQL 1.1 Update
  - SPARQL 1.1 Service Description
  - SPARQL 1.1 Basic Federated Query
  - Sesame 2.6.3
  - Ganglia Integration

- **2012 Q2**
  - SPARQL 1.1 Update Extensions (solution sets, cache control)
  - SPARQL 1.1 Property paths
  - New search index (subject centric)
  - Runtime query optimizer (RTO) for analytic queries
  - RDF “GOM”

- **2012 Q3**
  - High Availability for the RWStore

- **2012 Q4** – Scale-Out enhancements
  - Cloud-enabled storage management

See https://sourceforge.net/apps/mediawiki/bigdata/index.php?title=Roadmap
Runtime Query Optimization
RDF Join Order Optimization

- Typical approach
  - Assign estimated *cardinality* to each triple pattern.
    - Bigdata uses the fast range counts
  - Start with the most selective triple pattern
  - For each remaining join
    - Propagate variable bindings
    - Re-estimate cardinality of remaining joins

- Cardinality estimation error
  - Grows *exponentially* with join path length
Runtime Query Optimizer (RTO)

- Inspired by ROX (Runtime Optimization for XQuery)
  - Online, incremental and adaptive dynamic programming.
  - Breadth first estimation of actual cost of join paths.
  - Cost function is cumulative intermediate cardinality.
  - Extensions for SPARQL semantics.

- Query plans optimized in the data for each query.
  - Never produces a bad query plan.
  - Does not rely on statistical summaries.
  - Recognizes correlations in the data for the specific query.
  - Can improve the running time for some queries by 10x - 1000x.
  - Maximum payoff for high volume analytic queries.


See “ROX: Runtime Optimization of XQueries” by Riham Abdel Kader and Peter Boncz [http://oai.cwi.nl/oai/asset/14193/14193B.pdf]

Join Graphs

- Model set of joins as vertices
  - Connected by join variables
  - Implicit connections if variables shared through filters.
- Requires an index for each join
  - Easily satisfied for RDF.
- Starting vertices
  - Determined by their range counts.
Join Graph for LUBM Q9

SELECT ?x ?y ?z
WHERE {
  ?x a ub:Student . # v0
  ?y a ub:Faculty . # v1
  ?z a ub:Course . # v2
  ?x ub:advisor ?y . # v3
  ?y ub:teacherOf ?z . # v4
  ?x ub:takesCourse ?z . # v5
}

- 6 Vertices
- 3 Variables, each in 3 vertices.
- No filters.
Annotate with Range Counts

- Range count each vertex.
- Minimum cardinality for (1,4,2).
- Will begin with those vertices.

Note: We begin with vertices which are close to the minimum cardinality, not just the vertex with the minimum cardinality. This can avoid some local minima.
Incremental Join Path Estimation

- Evaluation proceeds in rounds.
- Increase sample size in each round.
  - Resample to reduce sampling bias.
  - Estimation error reduced as sample size increases.
- Estimate join cardinality using cut off joins
  - Push random sample of $N$ tuples into each vertex
  - Join “cut off” when $N$ tuples are output (handles overflow).
  - Output cardinality estimate can underflow (raise $N$ in next round).
- Join paths pruned when dominated by another path for the same vertices.

Round 0
Begin with minimum cardinality vertices $\{1,4\}$
Estimate join cardinality for one-step joins from those vertices:

$$\Rightarrow \{[1 \ 4], \ [4 \ 2]\}$$

Round 1: 2 paths in, 6 considered, 6 out.

$$\Rightarrow \{[1 \ 4 \ 2], \ [1 \ 4 \ 3], \ [1 \ 4 \ 5], \ [4 \ 2 \ 1], \ [4 \ 2 \ 3], \ [4 \ 2 \ 5]\}$$

Round 2: 6 paths in, 16 considered, 8 out.

$$\Rightarrow \{[1 \ 4 \ 3 \ 0], \ [1 \ 4 \ 3 \ 5], \ [1 \ 4 \ 5 \ 0], \ [4 \ 2 \ 1 \ 3], \ [4 \ 2 \ 3 \ 0], \ [4 \ 2 \ 3 \ 5], \ [4 \ 2 \ 5 \ 0]\}$$

Round 3: 8 paths in, 16 considered, 5 out.

$$\Rightarrow \{[1 \ 4 \ 3 \ 0 \ 2], \ [1 \ 4 \ 3 \ 5 \ 0], \ [1 \ 4 \ 3 \ 5 \ 2], \ [4 \ 2 \ 1 \ 5 \ 0], \ [4 \ 2 \ 3 \ 5 \ 0]\}$$

Round 4: 5 paths in, 5 considered, 1 out.

$$\Rightarrow [1 \ 4 \ 3 \ 5 \ 0 \ 2]$$
Round 0

Path: sumEstCard
- [1 4]: 1400
- [4 2]: 1627

?x a Student
?y a ub:Faculty
?z a ub:Course
?x ub:advisor ?y
?y ub:teacherOf ?z
?x ub:takesCourse ?z
Round 1

Path: sumEstCard
- [1 4 2]: 1500
- [1 4 3]: 1500
- [1 4 5]: 1500
- [4 2 1]: 3254
- [4 2 3]: 10421
- [4 2 5]: 21964

?x a Student
?y a ub:Faculty
?z a ub:Course

?x ub:advisor ?y
?y ub:teacherOf ?z
?x ub:takesCourse ?z
Round 2

Path: sumEstCard
- [1 4 3 0]: 8373
- [1 4 3 5]: 1686
- [1 4 5 0]: 44166
- [4 2 1 3]: 12685
- [4 2 1 5]: 30370
- [4 2 3 0]: 15639
- [4 2 3 5]: 10578
- [4 2 5 0]: 49080

6463
v0

?x a Student

v1

?y a ub:Faculty

v2

1627
v3

?x ub:advisor ?y

3191
v4

?y ub:teacherOf z

1627
v5

?z a ub:Course

21489

?x ub:takesCourse z
Round 3

Path: sumEstCard
- [1 4 3 0 2]: 14440
- [1 4 3 5 0]: 1791
- [1 4 3 5 2]: 1857
- [4 2 1 5 0]: 71045
- [4 2 3 5 0]: 10704

?x a Student
?x ub:advisor ?y

?y a ub:Faculty
?y ub:teacherOf ?z

?z a ub:Course
?x ub:takesCourse ?z
Round 4

Path: sumEstCard
- [1 4 3 5 0 2]: 1896

v0

v3

v1

v4

v2

v5

?x a Student

?x ub:advisor ?y

?y a ub:Faculty

?y ub:teacherOf ?z

?z a ub:Course

?x ub:takesCourse ?z

6463

3101

540

1627

1627

21489

540

1627

21489
### Selected join path

#### Table:

<table>
<thead>
<tr>
<th>vertex</th>
<th>srcCard</th>
<th>join ratio</th>
<th>in</th>
<th>limit</th>
<th>out</th>
<th>sumEstCard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>540E</td>
<td>2.97</td>
<td>800</td>
<td>337</td>
<td>1000</td>
<td>540</td>
</tr>
<tr>
<td>3</td>
<td>1602E</td>
<td>0.82</td>
<td>151</td>
<td>1000</td>
<td>1000</td>
<td>21752</td>
</tr>
<tr>
<td>5</td>
<td>1069E</td>
<td>0.02</td>
<td>700</td>
<td>1000</td>
<td>11</td>
<td>12917</td>
</tr>
<tr>
<td>0</td>
<td>166</td>
<td>0.54</td>
<td>11</td>
<td>1000</td>
<td>7</td>
<td>13022</td>
</tr>
<tr>
<td>2</td>
<td>105</td>
<td>1</td>
<td>7</td>
<td>1000</td>
<td>7</td>
<td>13127</td>
</tr>
</tbody>
</table>

75% Faster than the join path chosen by the static join order optimizer.

- **vertex**: The vertices in the join path in the selected evaluation order.
- **srcCard**: The estimated input cardinality to each join (E means Exact).
- **ratio**: The estimated join hit ratio for each join.
- **in**: The #of input solutions for each cutoff join.
- **limit**: The sample size for each cutoff join.
- **out**: The #of output solutions for each cutoff join.
- **sumEstCard**: The cumulative estimated cardinality of the join path.
Sub-Groups, Sub-Select, etc.

- Vectored left-to-right evaluation
- Solutions flowing into a group depend on join order in the parent

```
SELECT ?x ?y ?z WHERE {
  ?y a ub:Faculty .           # v1
  ?z a ub:Course .            # v2
  ?x ub:advisor ?y .          # v3
  ?y ub:teacherOf ?z .        # v4
  {   ?x a ub:Student .       # v0
      ?x ub:takesCourse ?z . # v5
  } | 
```

- Flatten into join graph
- Path extensions MUST respect SPARQL group boundaries
- Sample join path with groups:
  
  $1 \Rightarrow 4 \Rightarrow 3 \Rightarrow 5 \Rightarrow 0 \Rightarrow 2$
New Cloud Architecture

- Hybrid shared nothing / shared disk architecture
  - Compute cluster
    - Spin compute nodes up or down as required
      - plus
  - Managed cloud storage layer
    - S3, openstack, parallel file system, etc
- Global write-once contract
  - Writes replicated across a quorum using local disk.
  - Read only files migrated to storage layer
    - Old journals
    - New index segments
  - Compute nodes cache hot shards.
Bigdata®

RDF / GOM
Generic Object Model

- Flexible OODBMS
  - Developed by CTC Technologies since 1990s.
  - Data extensible
    - Objects are persistent property sets.
    - Zero or more schemas per object
  - Scalable collections.
  - Primary (object sets) and secondary (attribute sets) indices.
  - Transactional isolation provides consistency.

See the CTC GOM introduction paper: http://ctc-tech.biz/cutthecrap/software/whitepapers/theproject.html. There are also whitepaper links at the bottom of the page.

The GPO in five figures website section may also be useful: http://www.ctc-tech.biz/cutthecrap/software/gpo/fivefigures.html. This presents a high level overview of how to interact with the CTC GPO model.
GOM Example

- Generic objects may have any properties.
- Each object may be a member of zero or more classes.
- Each “class” has a schema and its instances are validated.

Person.name := “Bryan”
Person.dob := 1/13/1964
Person.sex := M
Employee.id = 3818
Employee.hire = 10/13/04
Employee.state = DC
Collections

- Forward references define collections.

```
<table>
<thead>
<tr>
<th>Employee</th>
<th>Person</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>first</td>
<td>dob</td>
<td>EN</td>
</tr>
<tr>
<td>state</td>
<td>sex</td>
<td></td>
</tr>
<tr>
<td>company</td>
<td></td>
<td>company</td>
</tr>
</tbody>
</table>
```

- “Bryan”
- “Mike”
- “SYSTAP”
Indices

- Collections may be indexed on any property (or computed property).

```
getIndex("company","name");
```

```
“Bryan”
```

```
“Mike”
```

```
“SYSTAP”
```
Many-to-Many Pattern

• Based on an intermediate object.

• Iterator patterns can expand and filter complex paths.
RDF “GOM”

- Harmonizes GOM with RDF
  - But Statements are also Resources (provenance).
- Clients interacts with object manager
  - Local operations against in memory model
  - Remote operations translated into efficient database queries
  - SPARQL used to materialize or grow interesting object graphs.
    - DESCRIBE ... WHERE {...}
- Object manager:
  - Maintains edit set.
  - Incrementally materialize graphs
- Code generation for multiple language bindings
  - Objects are data, “skinned” by different behaviors.
    - Java
    - JSON/JavaScript
    - ...
  - RDF “Alchemist” manages object models and code generation.
RDF with Provenance

RDF Object Model, extended to support provenance.

Supports provenance statements about (data|nodes) and link properties.
Comparison with Blueprints

In IGCM, the GPO is the Vertex and links do not have properties.

In igraph, applications do not concern themselves with indices.

Every vertex and edge in a GPO (edges have links).

Property names are fixed, property values are EQL literals.
GOM API / SPARQL Queries

/** Return first Value for property or null if no Value for that property. */
Value getValue(URI property);

/** Replace (self,p,?x) with (self,p,newValue). */
void setValue(URI property, Value newValue);

/** All (self,?p,?o). */
Set<Statement> getStatements();

/** All ?y where (self,p,?y). */
Set<Value> getValues(URI property);

/** Exists (self,p,?y). */
boolean isBound(URI property);

Object manager
Encapsulates all SPARQL requests
Edit set sent to server at commit.

Objects materialized using DESCRIBE ... WHERE ...

DESCRIBE cache in front of server.
Fast materialization.
Write through cache.

/** All ?y where (?y,/p,self). */
Set<IGPO> getLinksIn();

/** All ?y where (?y,?,self). */
ILinkSet getLinksOut(URI property);

/** All ?y where (self,?,?y) and ?y is Resource. */
Set<IGPO> getLinksOut();

/** All ?y where (self,?,?y) and ?y is Resource. */
ILinkSet getLinksOut(URI property);

/** Exists (?x,linkSet.property.self). */
boolean isMemberOf(ILinkSet linkSet);

/** * Return a map giving the range count for each reverse link property. */
* <pre>
* SELECT ?p, COUNT(*) WHERE { ?o ?p <s> }
* GROUP BY ?p
* </pre>
* Map<URI,Long> getReverseLinkProperties();
RDF “GOM” Object Model
Property Sets and Link Sets

- **Set**: Interface
  - get(name: URL) : Value
  - get(name: URL, val: Value)
  - getStatements() : Set<Statement>
  - getValues(property: URL) : Set<Value>
  - getInValues(property: URL) : Set<Value>
  - getInLinkSet(property: URL) : LinkSet
  - getOutLinkSet(property: URL) : LinkSet
  - inBound(property: URL) : boolean
  - isMemberOf(LinkSet) : boolean

- **LinkSet**: Interface
  - property: URL
  - getInSet(property: URL) : LinkSet
  - getOutSet(property: URL) : LinkSet
  - isCyclic() : boolean

SYSTAP™, LLC
© 2006-2012 All Rights Reserved
http://www.bigdata.com/blog
Bigdata® Use Cases

Some Examples
KaBOB (Knowledge Base of Biology)

Hunter Lab
University of Colorado
• Kevin Livingston
  – kevin.livingston@ucdenver.edu
• Mike Bada
• Bill Baumgartner
• Yuriy Malenkiy
• Larry Hunter (PI)

Open Biomedical Ontologies
Motivation for KaBOB

- Biomedical researchers face a profound challenge in keeping track and making sense of
  - numerous curated databases
  - rapidly growing literature
  - data from high-throughput experiments
- Semantic integration is necessary to effectively unify and reason over all this information
Knowledge Base of Biology (KaBOB)

17 databases
- Entrez Gene
- DIP
- UniProt
- GOA
- GAD
- HGNC
- InterPro

12 ontologies
- Gene Ontology
- Sequence Ontology
- Cell Type Ontology
- ChEBI
- NCBI Taxonomy
- Protein Ontology

Open Biomedical Ontologies

biomedical data & information
biomedical knowledge
application data

127
bigdata®
Presents at Graph Data Management 2012
http://www.bigdata.com/blog
KaBOB Current State

- Unified representations under Open Biomedical Ontologies (OBO)
- 8.7 Billion triples and counting
  - Database records
  - Unified biomedical entities (gene, proteins, etc.)
- Used to drive applications in
  - NLP / Text Mining
  - Visualization
  - Reasoning
- Working toward more and more unified biomedical representation beyond entities
  (protein-protein interactions, pathways, drug targets, ...)

bigdata®
Presented at Graph Data Management 2012
http://www.bigdata.com/blog
• High-performance knowledge discovery & collaboration web application
• Used by Pharmaceuticals, Biotechnology Companies, Healthcare Organizations
• A preferred standards compliant triple store is bigdata®
Manufacturing Product Data is Heterogeneous

…and difficult to find, re-use, and share
Inforbix Product Data Applications

- Using RDF and bigdata® to tackle the manufacturing data problem
- Federate and index disparate product data sources
- Fast Google-style search experience promoting re-use of high-value product data
bigdata®

Flexible
Reliable
Affordable
Web-scale computing.