KiKiDB:

a triplet store with two-key indexing

Joe Presbrey
presbrey@mit.edu

6.033 Design Project 1
Prof. Steve Ward TR2-3
TA. David Schultz
March 22, 2007

1. Introduction

A triplet is a three field data structure represented abstractly as (*subject* String, *relationship* String, *object* String). Efficient triplet storage is essential for effectively browsing large relational databases such as those in E-commerce applications[1] or the RDF graph data of the Semantic Web[2]. Applications of triplet stores typically have many subjects with related objects and far fewer types of relationships. Users of these applications are not usually interested in accessing a single triplet at a time. Instead, they want to access a collection of properties about a given object. Searching against just two of the fields for all matching triplets is the most common operation. This triplet store design is optimized for performance for this type of search.

KiKiDB is a two-key indexed triplet store designed to excel in environments where disk space is cheap and low latency searches are required. A triplet stored in the system is replicated in three hash tables uniquely indexed for efficient two-key searches.

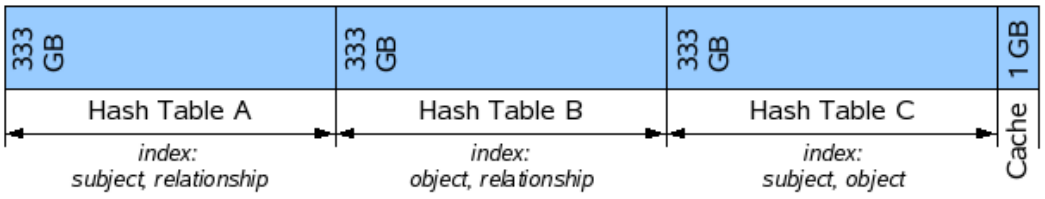


Figure 1. a terabyte physical disk partitioned for the KiKiDB triplet storage system

Each hash table is indexed with two different triplet fields as shown in Figure 1. The cache partition at the end of the disk stores RAM cache objects during system shutdown.

KiKiDB is desirable when disk space is inexpensive and search patterns are known to frequently take advantage of the system's two-key indexes.

2. Design Description

KiKiDB runs as a network server with RPC calls for manipulating the storage system. Section 2.1 presents the RPC interfaces, 2.2 describes the storage system's disk structures, and 2.3 describes the implementation of the RPC API calls.

2.1. Storage System Interface

This design supports an RPC interface with four calls for manipulating the triplet store. Insertion and deletion are named *insert()* and *remove()*. Their parameters are both similar to triplet form (*subject* String, *relationship* String, *object* String) and immediately affect the success of subsequent lookups. Lookups and searches are handled by *find()*. *find()* has 5 parameters (*subject* String, *relationship* String, *object* String, *start* integer, *count* integer). These parameters include the triplet fields like *insert()* and *remove()* except any of *find()*'s three string parameters also support the wildcard character ("*") which instructs the server to match all triplets regardless of the "wildcarded" field's contents. The *start* and *count* parameters are helpful range settings for returning a portion of a search with many results. *find(x,y,*,S,C)* skips S triplets and then returns no more than C subsequent triplets.

2.2. Disk Structures

Three hash tables are allocated on disk as shown in Figure 1. Hash tables contain triplets and index markers. Both of these are 128 bytes structures with two header status bits. The two status bits mark a sector when deleted and define it as a triplet or index marker accordingly. The triplet, index marker, hash table and cache partition disk structures are detailed in sections 2.2.1-2.2.4. Notes on garbage collection follow in Section 2.2.5.

2.2.1. Triplets

A triplet is a 128 byte sector stored at a hashed location in each of the three hash tables. Its data starts with two state bits followed by three null-terminated ("\0") strings as shown in Figure 2. If more space is necessary to store the strings, the sector may occupy two sequential sectors and indicate presence of the second sector by omitting the null-terminator in the first sector (see Figure 3).

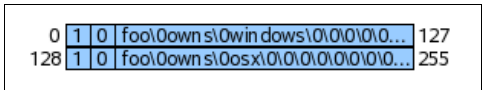


Figure 2. Two active triplets in the system: (bob, owns, windows)
(bob, owns, osx)

* *addresses are disk locations in bytes*

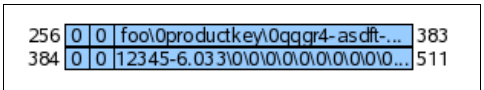


Figure 3. A deleted triplet spanning two 128-byte sectors:
(bob, productkey, qqgr4-asdft-...)

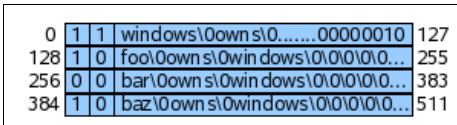
Though the definition of status bits could vary between different implementations, this paper presents with a scheme consistent with Figure 4.

	Active	Deleted
Triplets	1 0	0 0
Index Markers	1 1	0 1

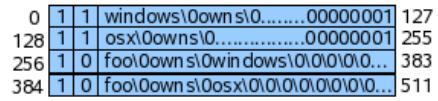
Figure 4. Status bit definitions

2.2.2. Index Markers

An index marker demarcates triplet sector groupings in hash tables as shown in Figure 5. A marker stores only the two fields indexed by its parent hash table. A third field is a binary integer that counts how many triplets matching the two indexed fields exist after the index marker. This improves search performance by advertising an expected number of results to *find()*. A count of N indicates that *find()* should stop seeking when searching after finding N matching triplets and return the results. An index marker is created when an inserted triplet has values for the table's indexed fields that do not have a preexisting index marker as shown in Figure 6. Index markers are also created by searches that returns no results to expedite future searches. Marker counts are incremented on *insert()*, decremented on *remove()*, and must precede all counted triplets.



The marker above (windows, owns, 2) is two because one of the three matching triplets was deleted.



Markers may be out of order due to order of inserts, deletes, and finds or hash collisions yet still improve performance.

Figure 5. Two possible sequences of markers and triplets from hash table B of Figure 1.

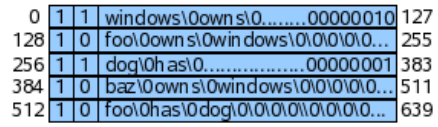


Figure 6. Inserting a triplet:
(1, 0, foo, has, dog)
to the 1st table of Figure 5 requires inserting a marker:
(1, 1, dog, has, 1)

The new index marker is inserted at 256 to replace the previously deleted sector and the new triplet is added at 512, the next empty sector after its marker.

2.2.3. Hash Tables

A hash table is formed of 4 KB disk blocks. Each block contains 32 active, deleted, or empty 128 byte triplet or marker sectors. The table location is obtained by hashing the two triplet fields it indexes. As shown in Figure 7, a function for hash table B in Figure 1 has the form hash(*object* String, *relationship* String). When a hash collision occurs during

insert(), the system references the disk usage bitmap[3] to locate the next empty sector using linear probing[4]. When placing an index marker sector, the next deleted or empty sector after the hashed location for the index marker is selected. When placing a triplet sector, the next deleted or empty sector after its index marker is selected. If an empty sector is reached in the seek for the index marker, a new index marker must be inserted as discussed in Section 2.2.2. The new index marker is placed before inserting the triplet as close after the hashed location for the triplet as possible as shown in Figure 6.

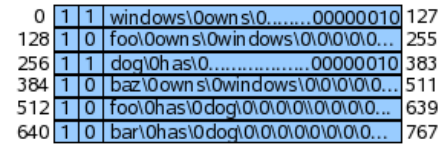


Figure 7. Possible hash results for table B in Figure 1:

hash(windows, owns) = 0*
hash(dog, has) = 256*
hash(monkey, has) = 512*
hash(osx, owns) = 768*

** These are unrealistic hashes. A good hash function for a 333 GB table would produce hashes at large increments*

2.2.4. Cache Partition

This space as shown in Figure 1 is not accessed during system runtime but contains two important structures utilized in RAM during operation and saved to this partition during *shutdown()*. The disk usage bitmap is a bit array setting bits for full blocks on the disk and clearing bits for blocks with free space (see Figure 8). The list of relationships contains all relationships that have been referenced by triplets in the system. It demarcates members with a null character as in Figure 9. The system performs one-key searches such as *find(subject,*,*)* and *find(*,*,object)* in an optimized fashion by converting them to two-key searches using all possible relationships from the list.

00000001000000000000000000000010...

1024*1024*1024*1024/4096/8 = 32 MB

Figure 8. An example disk usage bitmap for a mostly empty disk and the calculation for its size (for a 1 TB disk with 4 KB blocks)

owns\0has\0is\0...

Figure 9. An example relationship list

2.2.5. Garbage Collection

A long-running, active KiKiDB server accumulates many deleted sectors and zero-count index markers. Especially when the hash tables are nearly full, periodic garbage collection

during off-peak hours improves search efficiencies. When performing garbage collection, index marker and subsequent triplet sectors are shifted upwards replacing deleted and empty sectors in the hash table as close to their hashed location as possible. Since *find()* and other operations first seek to the hashed location of fields, performance is closely tied to how far index marker sectors are from their hashed locations. After sector shifts, the disk usage bitmap is updated to reflect changes in full disk blocks.

2.3. API Implementation

2.3.1. *insert(subject* String*, relationship* String*, object* String*)*

1. Perform the following steps for table A
    1. Hash the argument values of the indexed fields for the current table
    2. Call READ_BLOCKS on the hash location to find the index marker
    3. Find the next block with free sectors for writing in the disk usage bitmap if the current block is full
    4. Call WRITE_BLOCKS to increment the index marker or place a new index marker for this triplet as described in Section 2.2.3 and insert the triplet
        1. Write the new triplet to the same block if disk usage allows
        2. Call WRITE_BLOCKS again to insert the triplet in a free block if the index marker's block is full
    5. Mark the block in the disk usage bitmap as full if filled
2. Repeat step 1 for tables B and C
3. Append the relationship into the cached relationship list if its a new relationship

2.3.2. *remove(subject* String*, relationship* String*, object* String*)*

1. Perform the following steps for table A
    1. Hash the argument values of the indexed fields for the current table
    2. Call READ_BLOCKS on the hash location to find the index marker
    3. Call WRITE_BLOCKS to decrement the index marker
        1. If the triplet is on the same block, mark its deleted bit in the same write

2. Otherwise, call READ_BLOCKS to read {marker count/32} blocks in increments of free RAM to locate the triplet and or reach an empty sector
    1. If an empty sector is reached, the triplet does not exist
    2. If the triplet is finally located, call WRITE_BLOCKS to mark its deleted bit
4. Mark the block in the disk usage bitmap as no longer full

2. Repeat step 1 for tables B and C

2.3.3. *find(subject* String*, relationship* String*, object* String, *start* integer, *count* integer*)*

Case 1: one wildcard

1. Select and hash the argument values of a table that is not indexing the wildcarded field
2. Call READ_BLOCKS on the hash location to find the index marker and return the next (index marker count) number of matching triplets
3. Call READ_BLOCKS on {marker count/32} blocks if the index marker block does not contain all matches

Case 2: two wildcards except *find(*,relationship,*)*

Use case 1 with each relationship on the cached relationship list

Case 3: *find(*,relationship,*)*

Call READ_BLOCKS on an entire system defined table such as table A

Case 4: three wildcards

Call READ_BLOCKS on an entire system defined table such as table A

Case 5: no wildcards

1. Hash the argument values for the indexed fields of a system defined table such as table A
2. Call READ_BLOCKS on the hash location to find the index marker
    1. Return the triplet if its on the same block as the index marker
    2. Otherwise, call READ_BLOCKS to read up to {marker count/32} blocks in increments of free RAM to locate the triplet or reach an empty sector (triplet does not exist)

2.3.4. *shutdown()*

1. Garbage collect as described in Section 2.2.5
2. Remove old relationships from the relationship list and sort it
3. Save the contents of the RAM to the the cache partition

3. Design Analysis and Discussion

3.1. Design Decisions and Consequences

This design is most efficient when disk space is plentiful and search patterns are consistent with the design. This is the case for the two workload analyses described by this paper. A *find()* specifying any two triplet fields usually incurs only two seeks and a read per disk block of results. The less common three-key *find()* performs with equal performance. *insert()* and *remove()* are worse than *find()* because the system does not have pointers to exact triplets and must first perform the three-key *find()*. This trade-off is attractive when these operations occur less often than *find()* and the latency of these operations is not as important as search efficiency.

3.2. Workload Analysis

The following analyses assume one billion triplets are in the system with uniform data distrubution and limited table fragmentation. Disk seeks are required between reads and writes and writing is as fast as reading. In the best case, the manipulated triplet is on the same block as the index marker and is not in the worst.

3.2.1. Analysis of a photo-sharing application (Flickr^5++)

For this analysis, the following assumptions are made about the application to estimate system operation timings: 333 thousand users, 1 istype triplet/user, 10 albums/user, 1 istype triplet/album, 100 photos/album, 3 triplets/photo.

	Op.	Wait (ms)	Total	%	Weighted
insert() 1 triplet	Seek	73.02			
	R/W	0.36	73.38	0.05	3.67
insert()/remove() photo	Seek	219.06			
	R/W	1.08	220.14	0.05	11.01
find(,isa,monkey,0,1000)*	Seek	24.34			
	R/W	1.88	26.22	0.75	19.67
find(user,owns,,0,1000)*	Seek	24.34			
	R/W	1.88	26.22	0.15	3.93
Total Average Wait					38.27

Figure 10. Best-case analysis of the photo-sharing application.

The average wait time is 38.27 ms. Average throughput is 3.19 MB/s*.

For the find() calls alone, average throughput is 4.66 MB/s.

** for 1000 triplets at 128 bytes each*

	Op.	Wait (ms)	Total	%	Weighted
insert() 1 triplet	Seek	146.04			
	R/W	0.72	146.76	0.05	7.34
insert() photo	Seek	292.08			
	R/W	1.44	293.52	0.025	7.34
remove() photo	Seek	292.08			
	R/W	624378.2	624670.28	0.025	15616.76
find(,isa,monkey,0,1000)*	Seek	24.34			
	R/W	1.88	26.22	0.75	19.67
find(user,owns,,0,1000)*	Seek	24.34			
	R/W	1.88	26.22	0.15	3.93
Total Average Wait					15655.03

Figure 11. Worst-case analysis of the photo-sharing application.

The average wait time is about 16 seconds. Average throughput is down accordingly to 7.98 KB/s. Scanning table B for the API call remove(uri://myimage,istype,image) degrades the system significantly.

For the find() calls alone, average throughput is still 4.66 MB/s.

The photo-sharing application provides similar throughputs for best and worst cases since *find()* is the predominant operation and is such a consistently efficient operation in this design.

The efficiency of *insert()* is relatively consistent between best and worst cases since the hashes produced are either very specific (e.g., hash('uri://foouser', 'uri://fooalbum')) or very general (e.g., hash('image', 'istype')). When a hash is very specific, the system easily finds the location because there are so few. In the very general case, the system can use the index marker count to jump near or to the end of the duplicate triplets and append with minimal seeks.

In the worst-case, *remove()* significantly degrades system performance. A large number of photos (i.e., 333 million) match *find(*, istype, image)* and the system scans through all of them to find the matching triplet to delete.

3.2.2. Analysis of a library catalog application

For this analysis, the following assumptions are made about the application to calculate operation timings: 200 million books, 4 triplets/book, 50 million authors, 3 triplets/author, 4 books/author.

Loading 200 million books and 50 million authors would require inserting 950 million triplets. The 146.76 ms per-insert delay (see Figure 11) requires 54 months to insert all triplets. Since all operations of the workload are of the form *find(*,relationship,object)*, the library application can safely use a KiKiDB with only one hash table but this modified design still requires 17.93 months to insert all triplets. A batch insertion implementation is necessary for feasible bulk-loading of triplets.

	Op.	Wait (ms)	Total	Weight	Wait Weighted
find(*,title,bookname,0,1)	Seek	12.17		0.35=	
	R/W	0.06	12.23	0.35	4.28
find(*,name,authorname,0,1)	Seek	12.17		0.30+0.25=	
	R/W	0.06	12.23	0.55	6.73
find(*,author,uri://author,0,4)	Seek	12.17		0.30+0.25*100=	
	R/W	0.06	12.23	25.3	309.42
find(*,affiliation,affilname,0,100)	Seek	12.17		0.25=	
	R/W	0.19	12.36	0.25	3.09
find(*,publisher,pubname,0,1000)	Seek	12.17		0.10=	
	R/W	1.87	14.04	0.1	1.4
Weighted Wait					324.92
Total Weights				26.55	
Total Average Wait					**12.24**

Figure 12. Best-case analysis of the library catalog application

The average wait time is 12.24 ms. The lookup system is efficient.

The best-case scenario for the library application lookup system performs well. The average RPC wait is just barely greater than actual disk seek time as shown in Figure 12. This scenario assumes each institution is affiliated with 100 authors and publishers have 1000 books. In the worst-case when there are only 2 institutions for the 50 million authors and 2 publishers for the 200 million books, the average wait is about 1.16 seconds. This worst-case scenario is not very likely for this workload but demonstrates the impact of serving poorly indexed data.

	Op.	Wait (ms)	Total	Weight	Wait Weighted
find(*,title,bookname,0,1)	Seek	12.17		0.35=	
	R/W	0.06	12.23	0.35	4.28
find(*,name,authorname,0,1)	Seek	12.17		0.30+0.25=	
	R/W	0.06	12.23	0.55	6.73
find(*,author,uri://author,0,4)	Seek	12.17		0.30+0.25*100=	
	R/W	0.06	12.23	25.3	309.42
find(*,affiliation,affilname,0,25M)	Seek	12.17		0.25=	
	R/W	46875	46887.17	0.25	11721.79
find(*,publisher,pubname,0,100M)	Seek	12.17		0.10=	
	R/W	187500	187512.17	0.1	18751.22
Weighted Wait					30793.43
Total Weights				26.55	
Total Average Wait					**1159.83**

Figure 13. Worst-case analysis of the library catalog application

Poorly indexed data strains the server. This is not a likely scenario for an actual library catalog.

4. Conclusion

KiKiDB is a triplet database with high performance two-key searching. As storage becomes less and less expensive, storage systems and many other system designs will be willing

to sacrifice disk space for speed when necessary. KiKiDB is a good option for applications where this sacrifice is possible.

Because of the modularity of the design, KiKiDB systems are easily optimized. If an index is unnecessary for an application, I/O wait can be reduced across all API calls by dropping any of the three hash tables from the implementation. Hash table C indexing subject and object could be removed from future versions since *find(object, *, subject)* is rarely required for applications with a finite set of relationships.

Other possible modifications include adding a three-key indexed hash table to improve efficiency of *insert()* and *remove()* operations. A disk cache in RAM could also be useful for buffering writes to disk.

5. References and Acknowledgments

5.1. References

1. "Storage and Querying of E-Commerce Data," [Online document], 2001, [cited 2007 March 22]. Available HTTP: http://www.vldb.org/conf/2001/P149.pdf
2. "Semantic Web," [Online document], 2007 March, [cited 2007 March 22]. Available HTTP: http://esw.w3.org/topic/SemanticWebTools
3. "Bit array: Applications," [Online document], 2007 March, [cited 2007 March 22]. Available HTTP: http://en.wikipedia.org/wiki/Bit_array#Applications
4. "Linear probing," [Online document], 2007 March, [cited 2007 March 22]. Available HTTP: http://en.wikipedia.org/wiki/Linear_probing
5. "Flickr," [Online document], 2007 March, [cited 2007 March 22]. Available HTTP: http://www.flickr.com/

5.2. Acknowledgments

Thank you Leslie, Don, Mary, and the others from the writing department. You've provided comprehensive commentary on previous assignments and explicit examples of

expectations for this project.

Thank you David Schultz for staying after office hours giving feedback on this design and clarifying the goals of the assignment.

Thank you Steve Ward for making recitations consistently enjoyable with fantastic, humorous side-stories.

word count: 2529