© Copyright 2004, FlyMine, University of Cambridge

Improving Performance in Object Warehousing

Matthew Wakeling
FlyMine Group, Department of Genetics
University of Cambridge
matthew@flymine.org

© Copyright 2004, FlyMine, University of Cambridge

Introduction

- Most of what you will see in this talk is implemented by the FlyMine group, in the InterMine system (http://www.flymine.org/)
- This talk has two sections:
  - Techniques specific to data integration systems.
  - Techniques relevant to most throughput performance systems.
- It will help to have seen my previous talk, but most of this should make sense even if you haven't.

© Copyright 2004, FlyMine, University of Cambridge

Introduction - Coverage

- Large offsets in SQL.
- Writing to the database.
- Reading from the database (N + 1 reads, *etc.*).
- A three thread pipeline for data translation.
- Database-backed cached tables.
- Multi-CPU machines.
- Speeding up the DataLoader.
- Other tips.

© Copyright 2004, FlyMine, University of Cambridge

Large Offsets in SQL

- Consider an SQL query – for example:

```
SELECT fielda, fieldb FROM table ORDER BY fielda
```

  where there is an index on field `fielda`.

- If one pages the results in batches of 1000 by using LIMIT and OFFSET, and there are many rows, the SQL server will take time per batch proportional to the OFFSET.
- Therefore the entire results set will take time proportional to the square of the number of rows.

© Copyright 2004, FlyMine, University of Cambridge

Large Offsets in SQL

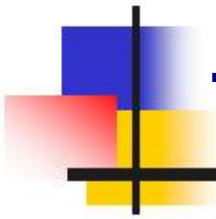
- To solve this:
  - When reading results, store some times where `fielda` changes from one value to another between one row and the next.
  - In subsequent queries, use this information to speed up the query, so if `fielda` changes from 42 to 45 between rows 98451 and 98452:

```
SELECT fielda, fieldb FROM table ORDER BY fielda OFFSET 100000
```

becomes:

```
SELECT fielda, fieldb FROM table WHERE fielda > 42 ORDER BY
      fielda OFFSET 1548
```

© Copyright 2004, FlyMine, University of Cambridge



The SQL Optimiser, Revisited

- One particular problem: some database servers have a limited notion of sort order.
  - Consider the query (generated by the optimiser):

```
SELECT precomp.a, precomp.b FROM precomp WHERE precomp.a > 42
   ORDER BY precomp.a, precomp.b OFFSET 1548
```

where the precomputed table is *already* sorted by field "a" then field "b".

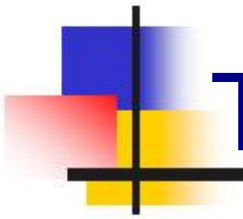
- You would expect this query to run really quickly.
- Actually, some databases only realise that the precomputed table is sorted by field "a".
- The database server will sort the whole table – **slow**.

© Copyright 2004, FlyMine, University of Cambridge

The SQL Optimiser, Revisited

- There are two things that the database server could do to fix this:
  - Have more sophisticated statistics, so (with the help of an ordered index) it can realise that it can efficiently retrieve the table in the correct order.
  - Implement a partial sorting algorithm, which would sort blocks of equivalent "a" value by "b" value.
    - These would be very useful anyway.

© Copyright 2004, FlyMine, University of Cambridge



The SQL Optimiser, Revisited

- If field “a” and “b” are integers, a workaround is possible:
  - Create the precomputed table like this:

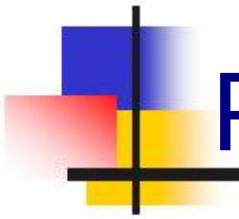
```
SELECT a, b, (a:numeric * 10000000000000000000) + b
  AS orderby_field FROM table ORDER BY a, b
```

Now orderby_field is guaranteed to have the same order as “a” then “b”. Create a B-tree index over orderby_field.

  - This optimised query will be fast, as the order and constraint are based on a sorted index:

```
SELECT a, b FROM precomp WHERE orderby_field >
  4299999999999999999 ORDER BY orderby_field OFFSET 1548
```

© Copyright 2004, FlyMine, University of Cambridge



Round-trip Time *vs* Bandwidth

- This is the underlying principle of much of this talk.
- If a system waits for a message round-trip time often, it will be slow, regardless of the bandwidth of the communication channel.
- This applies to databases, hard drives, RAM, system calls, network communications, *etc.*
- To avoid trouble, try to do a lot of work per request.

© Copyright 2004, FlyMine, University of Cambridge

Writing to the Database

- When storing new objects, they need to be assigned a unique ID – this number is normally generated by a sequence in the database.
- Don't access the database every time you need a new ID – only every 1000 times or so.

© Copyright 2004, FlyMine, University of Cambridge

Writing to the Database

- When writing inside a transaction, batch up all the individual writes and send them to the database in one large operation. The batch needs to be sent to the database:
  - If **you** wish to read from the same database,
  - When you commit the transaction,
  - If the batch gets too big.
- When flushing the batch, use the fastest available method to write many rows (*e.g.* prepared statements or COPY commands).

Colorado Software Summit

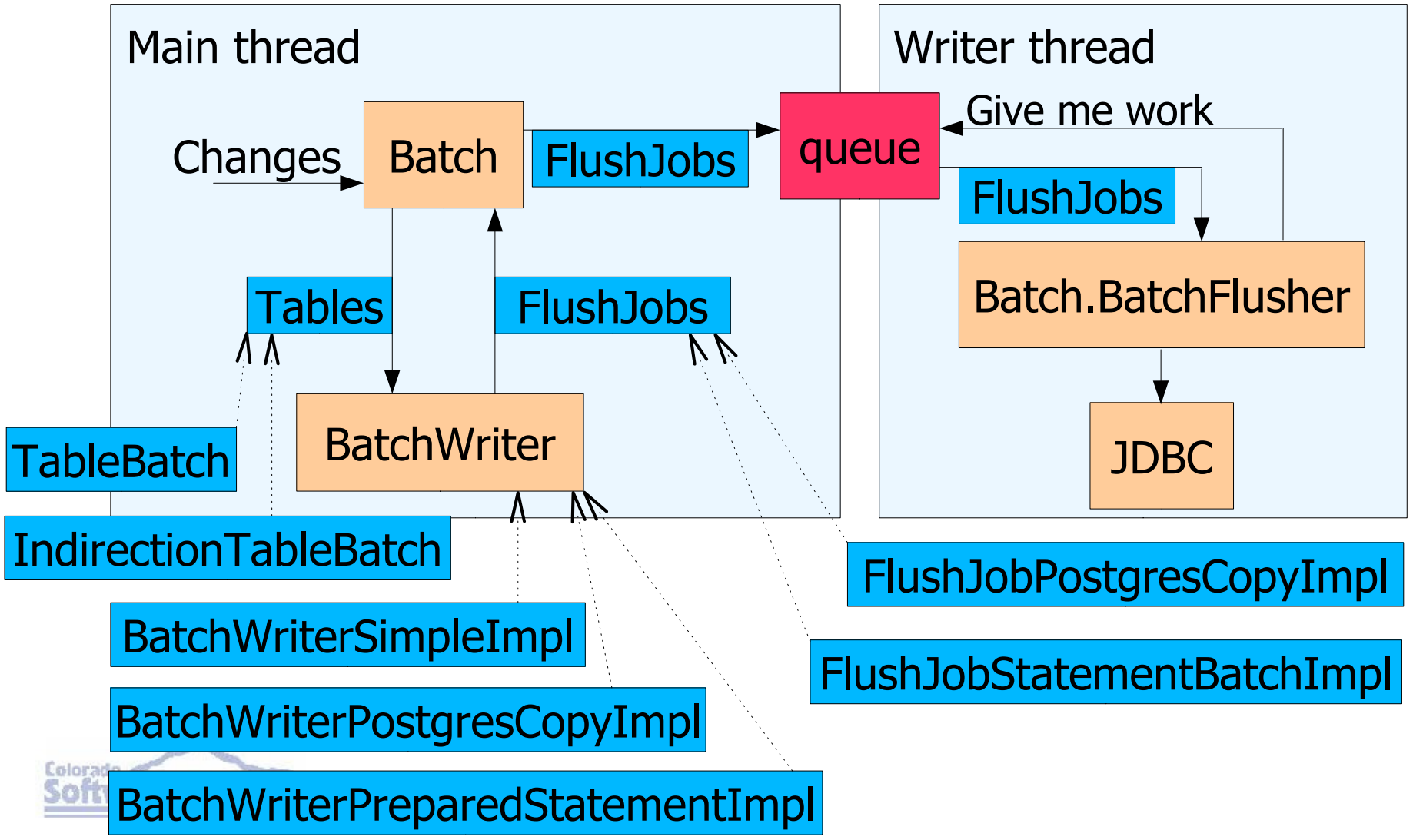
© Copyright 2004, FlyMine, University of Cambridge

Writing to the Database

- Perform writes in another thread.
  - That way your processing thread can keep working while the Database is busy storing data.
  - In fact, try to do as little processing as possible in the writer thread – any processing done consumes time where the Database is idle.
    - Create all your `java.sql.Statement` batches (and similar) in the processing thread, so the writer only has to call `statement.executeBatch()` (or similar).
  - However, make sure you properly flush the batch (and wait for the writer to finish) before committing or reading, so you notice errors.

© Copyright 2004, FlyMine, University of Cambridge

Writing to the Database
org.intermine.sql.writebatch



© Copyright 2004, FlyMine, University of Cambridge

Writing to the Database
org.intermine.sql.writebatch

```
public interface BatchWriter {
   public List<FlushJob> write(Connection con, Map
        tables) throws SQLException;
}
public interface FlushJob {
   public void flush() throws SQLException;
}
public class Batch {
   addRow(), deleteRow(), etc.
   public void clear();
   public void backgroundFlush() throws SQLException;
   public void flush() throws SQLException;
   public void close() throws SQLException;
   public void setBatchWriter(BatchWriter batchWriter);
   private synchronized void putFlushJobs
        (List<FlushJob> jobs) throws SQLExeption;
   private synchronized List<FlushJob> getFlushJobs();
}
```

© Copyright 2004, FlyMine, University of Cambridge

Writing to the Database

- Warning: The Batch will be accessing the Connection behind your back.
  - Calling flush() will guarantee that the Connection is unused until you call another Batch method.
- However: Each time you call flush(), you must wait for the entire batch to be flushed.
  - This may take several seconds. The processing thread will be idle, then the writing thread will be idle until the processing thread catches up.
  - For example, fetching new IDs for every 1000 objects written can be crippling.

© Copyright 2004, FlyMine, University of Cambridge

Writing to the Database

- What if someone calls flush() very frequently?
  - For example, before every read.
- Reduce the amount of data flushed.
  - Only bother flushing those tables that are about to be read – do a partial flush.
- If the flush is a NO-OP, don't even bother passing the empty job to the writer thread.
  - Saves thread context-switches.
  - Even so, make sure we wait for any unfinished writer activity, and pass back any exceptions.

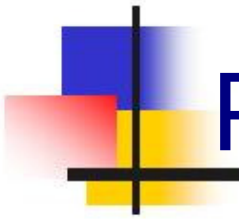
Colorado Software Summit

© Copyright 2004, FlyMine, University of Cambridge

Writing to the Database

- Deleting rows is expensive.
  - If you are merely filling a database, you can avoid it completely. That way, you can remove the indexes while building, then create the indexes afterwards.
  - Keep track of the IDs that have been created but not stored. You can guarantee that those do not already exist in the database, so when storing you do not need to delete first.
  - Use `SoftReference` – you don't want to fill up memory with IDs.

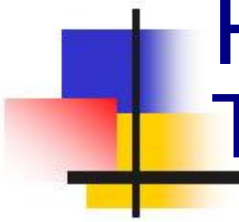
© Copyright 2004, FlyMine, University of Cambridge



Reading from the Database

- Batching – choose a sensible batch size
  - Bigger is faster, but only to a point.
  - A single batch must fit in memory.
  - 1000 to 5000 rows is sensible.
  - If you are paranoid about large rows fitting in memory, then run an SQL query to find out how big the rows are (*e.g.* use `char_length()`), and retrieve as many as required to fit in a certain amount of memory.
    - DirectDBReader

© Copyright 2004, FlyMine, University of Cambridge



Reading from the Database
The N + 1 Reads Problem

- Consider the case where you write a query returning objects. You iterate through the results, and call `toString()` on each object.
- The `toString()` method prints out information included in objects described by references from the object.
- Each reference followed causes a query to be run.
- Therefore you run N + 1 queries – it is slow.

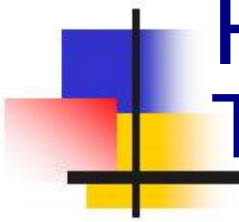
© Copyright 2004, FlyMine, University of Cambridge

Reading from the Database
The N + 1 Reads Problem

- Solution 1: Rewrite your query.
  - Tricky – you need LEFT OUTER JOIN to cope with the case where the reference is `null`.
  - This does not work well with collections.
- Solution 2: Make the proxies aware of each other, so reading one would cause the others to be fetched in a single batch read.
  - This could be done, but it leaves objects that are weirdly connected together somehow. Garbage collecting them may be interesting.

Colorado Software Summit

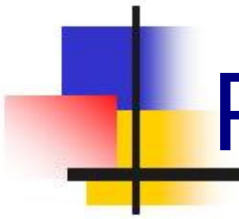
© Copyright 2004, FlyMine, University of Cambridge



Reading from the Database
The N + 1 Reads Problem

- Solution 3: Write an ObjectStore that loads all first-level collections and references.
  - ObjectStoreFastCollectionsImpl
- Solution 4: Give the ObjectStore explicit knowledge of what data is likely to be needed.
  - BatchingDBReader (DBConverter – builds a cache of likely SQL queries. When the DBConverter tries to run an SQL query, it is usually found in the cache)
  - ObjectStoreItemPathFollowingImpl (DataTranslator)
  - ObjectStoreFastCollectionsForTranslatorImpl (DataLoader)

© Copyright 2004, FlyMine, University of Cambridge



Reading from the Database

- Perform reads in another thread.
  - When your processing thread needs more data, it is immediately available.
  - Do as little processing in prefetch thread as possible, so the database isn't needlessly idle.
  - You can **guess** what data will be required next (like the PrefetchManager, which spots patterns of access in Results objects).
  - You can **know** what data will be required next (ReadAheadDBReader).

© Copyright 2004, FlyMine, University of Cambridge

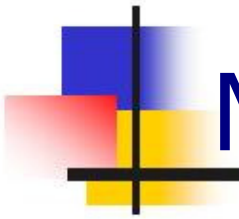
Three Thread Pipeline

- This is the ideal single-processor bulk transfer architecture.

  - Do as much processing as possible in a single thread. Reading and writing threads should only wait for the database or the processing thread.
  - This should mean that at least one of the three stages should be going at full speed.
- However, this does not work if the processing thread needs to search in the destination database.

Colorado Software Summit

© Copyright 2004, FlyMine, University of Cambridge



Multiprocessor Machines

- There is a huge overhead involved in moving data between two CPUs. Avoid.
  - This is particularly the case for Java, because objects are scattered around the heap.
  - Transferring an eight-byte object involves flushing and loading a 64 or 128 byte cache line, with maybe a thousand CPU cycles of latency, plus thread scheduling and synchronisation overheads.
  - An array of ints is faster than an array of Integers.
- If there is little data but much processing, then this is not a problem.

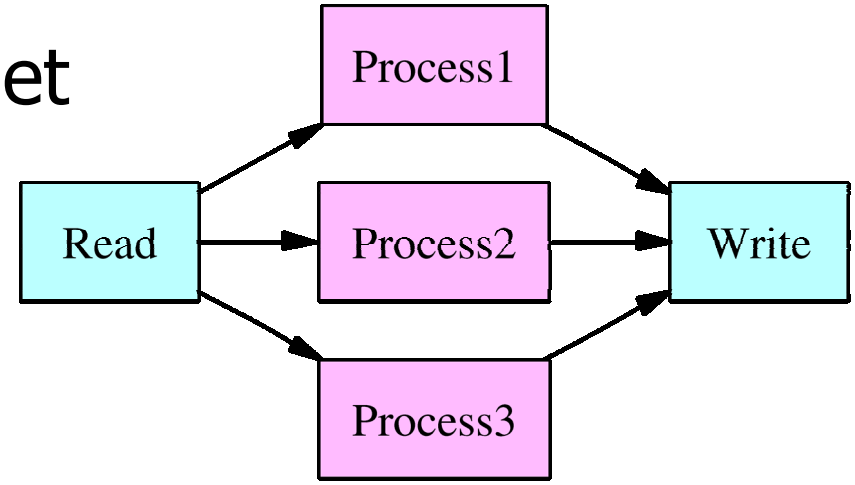
© Copyright 2004, FlyMine, University of Cambridge

Multiprocessor Machines

- In particular, avoid a thread model like this:

  - You are transferring data between threads (and probably CPUs) four times.

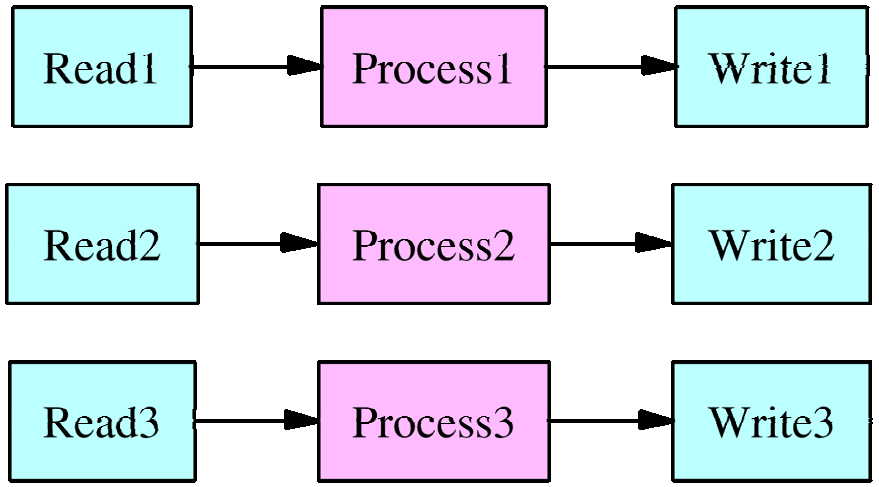
- You might be able to get away with this thread model, which would be reasonably easy to implement.



© Copyright 2004, FlyMine, University of Cambridge

Multiprocessor Machines

- This thread model might seem a good idea, however the threads in each group may be run on different processors.

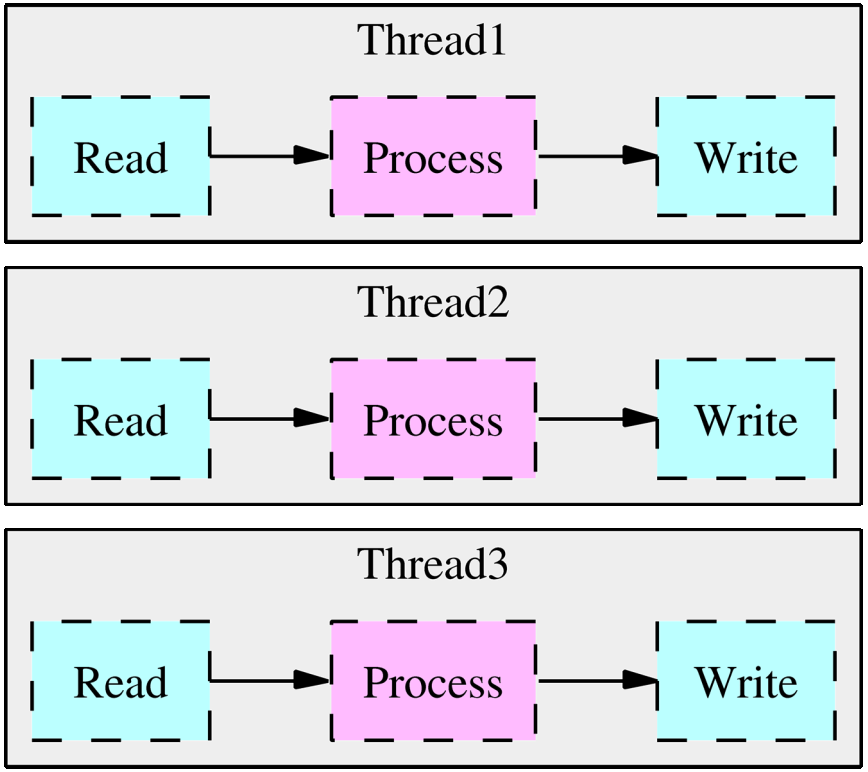


  - Data is still being transferred between CPUs.
  - This method is no better than the previous thread model.

© Copyright 2004, FlyMine, University of Cambridge

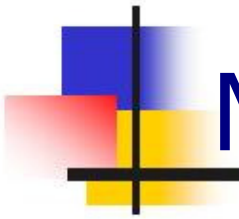
Multiprocessor Machines

- The solution is to use this thread model, where each thread takes a unit of work from start to finish. You will need to use non-blocking I/O in order to gain the benefits of the three thread model.



  - But how does one do non-blocking JDBC I/O?

© Copyright 2004, FlyMine, University of Cambridge



Multiprocessor Machines

- It might be the case that your process is I/O bound – so throwing more CPU power at it will never make it any faster.
- In any case, discs don't tend to respond to multiprocessing very well. Their best performing operation is sequential access.
  - Although some multiprocessing may help.
- This is a difficult topic, still being researched, so try whatever you can think of. Don't be surprised if it goes slower instead of faster when you try something.

Colorado Software Summit

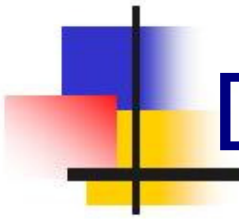
© Copyright 2004, FlyMine, University of Cambridge

Multiprocessor Machines

- If you really want to make it faster, then you will have to learn all about CPUs, caches, memory latency, *etc.*
  - It may not be worth the cost.
- If you can split the task up into several independent tasks, you could run it on several separate machines, and avoid these problems completely.
  - For example, if you need a fast static page web server, just use several identical machines, each with their own copy of the pages.

Colorado Software Summit

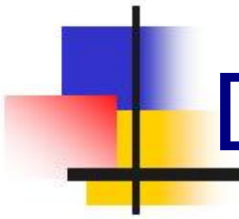
© Copyright 2004, FlyMine, University of Cambridge



Database-backed Lookup Table

- Aim: provide a lookup service, much like a `java.util.Map`, but for specific data types.
  - Allow random access for both reads and writes.
  - There is too much data to fit in RAM, so it must overflow into a database table.
  - We want a sizeable cache, so read operations don't have to hit the database very often.
  - We want to use bulk writes to the database, because they are faster.
  - We never want to wait for a write operation.

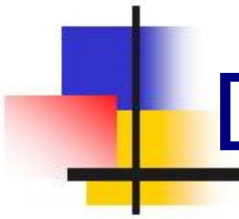
© Copyright 2004, FlyMine, University of Cambridge



Database-backed Lookup Table

- We want a Least-Recently-Used cache – `java.util.LinkedHashMap` is good for this.
  - However, it doesn't shrink and expand according to memory constraints, like SoftReference. You need to set a maximum number of entries.
- Writes always go straight into the cache.
  - You may need to perform a read first.
  - When the cache becomes too big, atomically separate out a least-recently-used portion.
  - This portion will then be written in a separate thread, over a dedicated "writing" connection.

© Copyright 2004, FlyMine, University of Cambridge



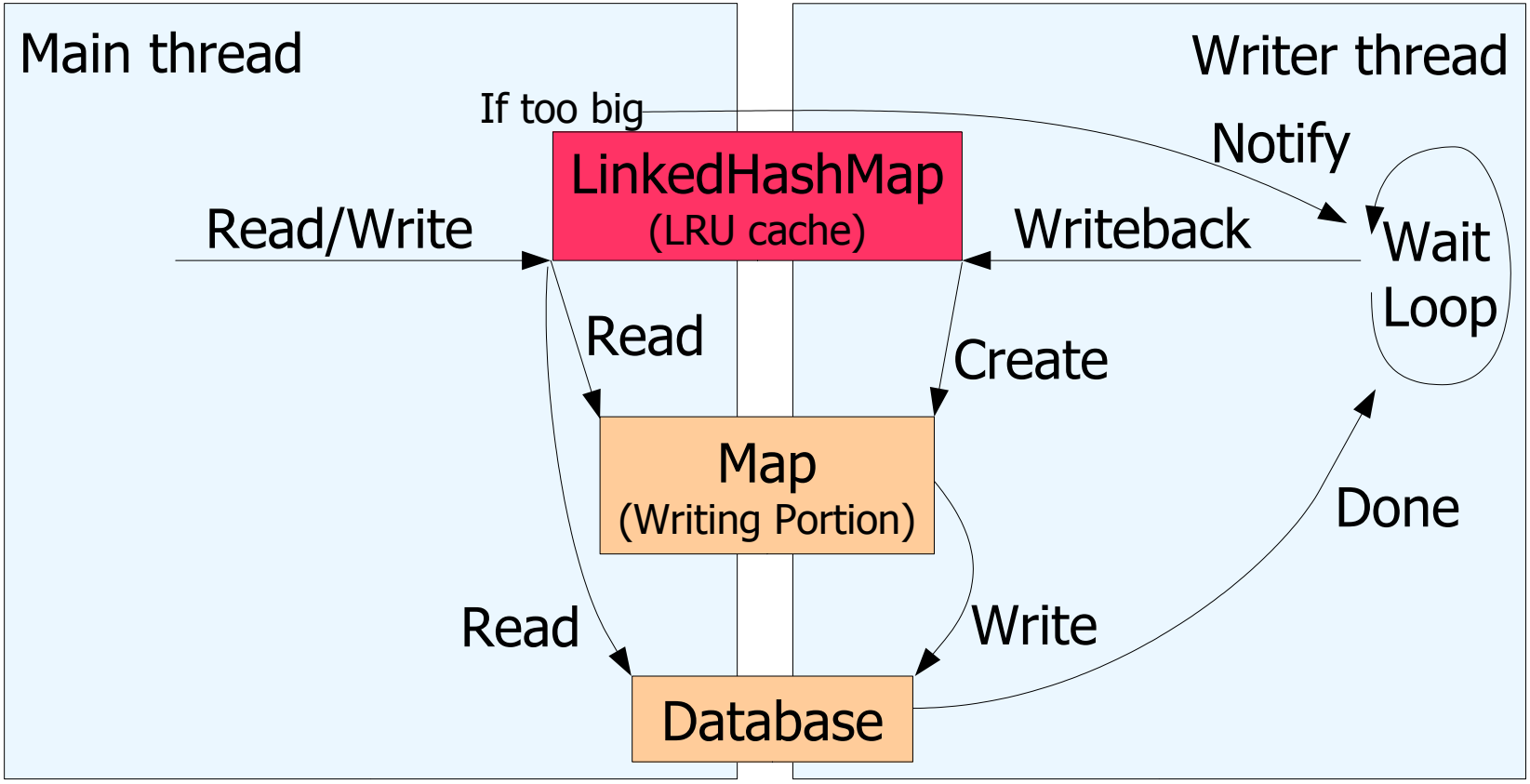
Database-backed Lookup Table

- If a read operation misses the cache, it must be fetched from the database.
  - It must return the same result regardless of what the writing thread is doing.
    - Therefore, you must look in the portion being written, and that portion must not be removed from memory until the database write has been committed.
  - It should put the entry into the cache for later.
- Warning: If entries are created faster than they can be written to the database, you must block the user.

Colorado Software Summit

© Copyright 2004, FlyMine, University of Cambridge

Database-backed Lookup Table



© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader

- The FlyMine database build process has three stages:
  - DataConverter – converts the format of data.
  - DataTranslator – converts the model of data.
  - DataLoader – integrates multiple overlapping data sources.
- The first two fitted well into the three-thread pipeline architecture, going from 200 objects per minute to 140,000 objects per minute.
- The DataLoader is not so simple.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader

- There are three reasons the DataLoader does not fit into the three-thread pipeline:
  - It has to run queries in the destination database, which may be affected by what was written in the previous operation. Therefore write batching cannot happen.
  - It has to run at least one query per object, where the other two stages can get away with one query for a thousand objects.
  - It needs a database-backed lookup table, which might need to run a query for each object.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader

- The overwhelming bottleneck is the query that is run on the destination database.
- In a naïve DataLoader implementation, that query may be run many times per object.
  - Once for the object itself, then once for all the objects referenced by it.
- This query searches for any objects in the destination database that are "equivalent" to the given object. The given object is then merged with its counterparts in the database.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Using an ID Map

- If we make the assumption that objects passed to the DataLoader have a consistent ID field set, we can avoid running the query more than once per object.
  - We can assume that we will always get the same result each time we run the query for an object with a certain ID.
    - This isn't quite true – the first time it is run, there may be zero, one, or many "equivalent" objects, and subsequent times there will always be one. However, we can work out what that one will be from our actions from the first time.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Using an ID Map

- Any time we run the query, we will always end up mapping one source object onto one destination object, and that mapping will be constant until the end of the DataLoader run.
  - These objects can be identified by ID.
  - Therefore, we can create a Map from ID to ID.
  - If IDs are Integers, one can use LinkedHashMap with Integers (~40 bytes per entry) and limit the size.
  - Alternatively, write an IntToIntMap (~4 bytes per entry), and don't bother limiting the size.

Colorado Software Summit

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Using an ID Map

- Having the ID Map helps because:
  - The DataLoader can perform a simple query to fetch the destination object by ID (which might hit the cache), rather than a complex search.
  - The DataLoader can also in some circumstances skip actually merging the objects (and therefore dealing with all the objects referred to by it).
  - For an object that is not in the ID map, if part of the search key includes a reference to an object which is in the ID map, that makes the query a lot simpler (and faster).

Colorado Software Summit

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Cheating with Fetching Objects

- However, caches need to be flushed every time a write happens, so fetching an object by ID is unlikely to actually hit the cache.
  - If an object has already been seen (*i.e.* it is in the ID map), then it is unlikely that it will actually be needed. Instead of fetching the object, one can return a ProxyReference, which is capable of fetching the object if required.
  - A ProxyReference is also a DBObject, so if an object is stored with a reference to a ProxyReference, it will end up with a reference to the correct object in the database.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Breaking Causality

- We still have the fundamental problem that we need to run a query for each object, that may depend on data only just written.
  - or does it?
  - Consider that the only thing that can alter the results of the equivalent objects query is storing that particular object, and even then the change is predictable.
  - We can make the assumption that we will never attempt to store an object that is equivalent to a previously-stored (in this data run) object that isn't recognised by the ID map.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Breaking Causality

- Having made that assumption, we can allow the equivalent object query to be performed well in advance of when it is needed.
  - Specifically, it can be performed on an ObjectStore outside transactions, while writes are being made to an ObjectStoreWriter inside a transaction.
  - This will speed up operations slightly, because writes can then batch up, and caches will be more coherent. This isn't actually much advantage, as the bottleneck is performing the queries.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Breaking Causality

- We could parallelise the equivalent objects data collection by using several threads and connections.
  - This probably won't help much, because it does not decrease the amount of work that the database server has to do. The disc drives will not be able to seek to the right place much quicker because it has a list of five things to do rather than just one.
  - We still need to be careful about objects affecting each other if we go multi-threaded.

Colorado Software Summit

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Breaking Causality

- We could speed up the equivalent objects data collection by combining several queries into one batched query.
  - This would be a **pain** to implement, because the results returned would contain objects equivalent to one of a set of objects that created the query. You would have to write code to allocate each returned object to the source object to which it is equivalent.
  - This probably will improve performance, because the database can use better search algorithms if it is given the choice.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Breaking Causality

- We can in fact run the equivalence queries on a snapshot of the destination database at the time immediately before we started loading this data source.
  - So, we could clone the destination database onto several servers, and farm out the equivalence queries to them.
  - This would multiply the performance by nearly the number of servers we can use, assuming the CPU can keep up, and the synchronisation allows that level of multi-processing.

© Copyright 2004, FlyMine, University of Cambridge

Speeding Up the DataLoader
Knowing When To Stop

- I have not implemented batching or parallelising equivalent object queries.
  - It would be tricky.
  - The current DataLoader maxes out a Pentium 4 Xeon 2.8GHz. There is no room for more speed without making the Java more CPU-efficient (which I did) or getting a much bigger machine.
  - The DataLoader now runs at 17,000 objects per minute (up from 12 objects per minute). At that rate, it can build the entire database in under a day. Spending a month coding to improve that doesn't make sense.

Colorado Software Summit

© Copyright 2004, FlyMine, University of Cambridge

Other Tips

- Get the database indexes right.
  - This can make a 1000-fold difference easily.
- Profile your program. Work out where it is spending its time, and then optimise that bit.
- Use StringBuffer, not String concatenation.
- Remove any unneccessary features that slow the system down.
  - For example, the database schema OBJECT field, and the DB Object table.

© Copyright 2004, FlyMine, University of Cambridge

Other Tips

- Throw a bigger computer at it.
  - But be careful – multiprocessor machines may not help that much.
  - If your task is I/O bound already, giving it a bigger CPU will not help. Give it a nice RAID array or gigabit ethernet instead.
- Beware of round-trip times. If your operation is disc-bound, and is constantly seeking, then replacing a 20MB/s 7ms seek disc with a 50MB/s 10ms seek disc will make it **slower**.

© Copyright 2004, FlyMine, University of Cambridge

Other Tips

- Be careful with SQL databases – if you choose a certain character encoding, you may find your queries suddenly go much slower. SQL_ASCII is safe on PostgreSQL.
- Moore's Law: Computers get twice as fast every 18 months.
  - So, say you have money to buy a good computer now that would do your task in six years. If you wait 18 months before buying the computer, you will finish in four and a half years instead.

© Copyright 2004, FlyMine, University of Cambridge

Acknowledgements

The FlyMine team: Richard Smith, Matthew Wakeling, Mark Woodbridge, François Guillier, Rachel Lyne, Kim Rutherford, Wenyan Ji, Tom Riley and Gos Micklem.

The InterMine system (www.intermine.org) is a generic object-oriented database and data integration system, open source and licensed under the LGPL.

FlyMine (www.flymine.org) is a biology-specific application of the InterMine system, also licensed under the LGPL.

FlyMine is funded by the Wellcome Trust (grant no. 067205), awarded to M. Ashburner, G. Micklem, S. Russell, K. Lilley, and K. Mizuguchi.

FlyMine Group, Department of Genetics, University of Cambridge, Downing Street, Cambridge, CB2 3EH, UK
Tel: +44 1223 333377 Email: info@flymine.org