Main Memory Databases / H-Store

Lab 3 Due Today
Quiz 1 Monday

Today we are going to talk about main memory databases -- where we assume that all (or the bulk of) data fits into memory.

Will focus specifically on the case of transaction processing, e.g., operational databases for banks, websites, etc.

Claim is that these are likely to be able to fit into memory. Why?

Because main memory sizes are growing like crazy (I can buy a server w/ 1TB of RAM for about $10K.)

Databases don't grow that fast -- Amazon only sells so many products; Bank of America only has some many bank accounts.

The vast majority of these kinds of transaction databases now easily fit into the memory of one or a small number of machines.

But we have all of this database architecture designed to assume the case where data primarily resides on disk.

What if we were to design the system differently? E.g., from the ground up assuming data would fit into memory?

- Wouldn't need a buffer pool
- Could use new index structures with pages optimized for RAM (e.g., cache-line sized)

How much would that get us?

To find out, we took a conventional database (an open source system called Shore) and stripped out pieces of code to see how fast it ran.

(Show slide)

Ok -- so buffer pool and B+Trees look like they are about 50% of what the database has to do.

Why are they so much?

Every time I access a record, I make a call to the buffer pool. Lots and lots and lots of
function calls, and hash table looks, and picking through page structures.

**How could I do better?**

Just store a tree in memory for each table, with leaves of tree containing in-memory pointers to actual records. No more having to parse or pick apart bytes out of page objects stored on disk. Lots less function calls.

What is all this other stuff? Is it worth rewriting my database for a (at most) 50% gain? Can I get rid of locking, latching, and logging too?

To understand this, we need to understand why this stuff exists.

Conventional databases were designed for high levels of concurrency (lots of transactions running at once.) Latching and locking is designed to deal with this.

Why do conventional systems use so much concurrency? To mask slow operations.

What kind of slow operations might we worry about?

1) Things that go to disk
2) User stalls
3) Long running transactions

Can we make these go away?

1) Yes, that's the point (although recovery still uses disk)

2) User stalls: not common in the real world. It used to be that humans actually wrote queries against these systems (e.g., travel agents with a database front end); these days transactions are issued by computers, and systems are designed to avoid stalls because they cause parts of the system to freeze up. E.g., Amazon's shopping cart isn't transactional.

Instead of letting people issue statement - think - statement - commit, change so that users have to issue "stored procedures" that contain all of their statements and interleaving app logic that is executed in a single shot.

3) Long running transactions -

So we have a system without slow operations -- what does that mean?

Well -- maybe we can just not have any concurrency and run one transaction at a time
to completion.

Is that a good idea?

Yes, on a single processor. We don't have to pay the costs to lock data structures before we access them.

But what about multiple processors?

This paper proposes **partitioning**. Split the database into a number of partitions, and arrange so that most transactions run on only one partitioning.

Is that feasible / realistic?

Sure, many workloads can partition. Consider banking, or an email server: vast majority of operations are on behalf of a single user to their account(s). So we can easily partition on user or account id.

In a retail setting could partition on store, or warehouse.

What about workloads that don't partition?

This technique won't work. The H-Store system includes tools that try to automate partitioning; mostly work but some workloads do better than others.

What about transactions that span partitions?

Of course these will occur (e.g., transfers from account A to account B). Simplest propose is to lock both partitions before accessing. Of course with lots of multi-partition transactions scalability will be bad.

What are some other problems with partitioning? Skew. Some users will place a ton of load on the system, but we can only employ one thread / partition for their queries.

We will talk later in the class about other ways we might avoid some of the overheads of locking at latching. But clearly this partitioning approach can eliminate them altogether.

What about recovery?

Recovery is the process that ensures that databases are durable. Conventional design -- that we will study in a few weeks -- involves writing a log record per update, plus on every transaction commit. This I/O can be done sequentially -- so can be pretty fast -- but still isn't totally free.
Represents about 12% of transaction time in Shore (to generate and enqueue a log record for each transaction, that includes before and after state, etc.)

What does this paper propose as an alternative?

Replication: issue each transaction to several replicas, so that in the event of a crash there is another replica available to recover from.

Making this work is tricky. Paper proposes waiting for a certain amount of time before committing each transaction at a replica, to account for any possible network delays, but this makes some strong assumptions about performance of networks, and introduces latency that may or may not affect transaction throughput.

We'll revisit the question of how to implement replication, and approaches to it, later.

Ok, so how well does this work?

TPC-C -- good to be familiar with it

![Figure 1: TPC-C Schema (reproduced from the TPC-C specification version 5.8.0, page 10)](image)

Queries do things like issue a new order, process a payment, compute the stock levels for a given product, etc.

By partitioning on warehouse can get most transactions to go to a single site.

70,416 transactions per second, vs 850 on Oracle

Sounds pretty good!

Study break -- review PS2 part 2
Next week we are going to start to talk about how conventional transaction processing works, looking at the canonical design (which this paper is claiming is dead), and then at several other ways to make a transactional system go fast.

We'll revisit some of these transactional design decisions then.