- HBase performance improvements
- Obscure HBase features

salesforce.com

About Me

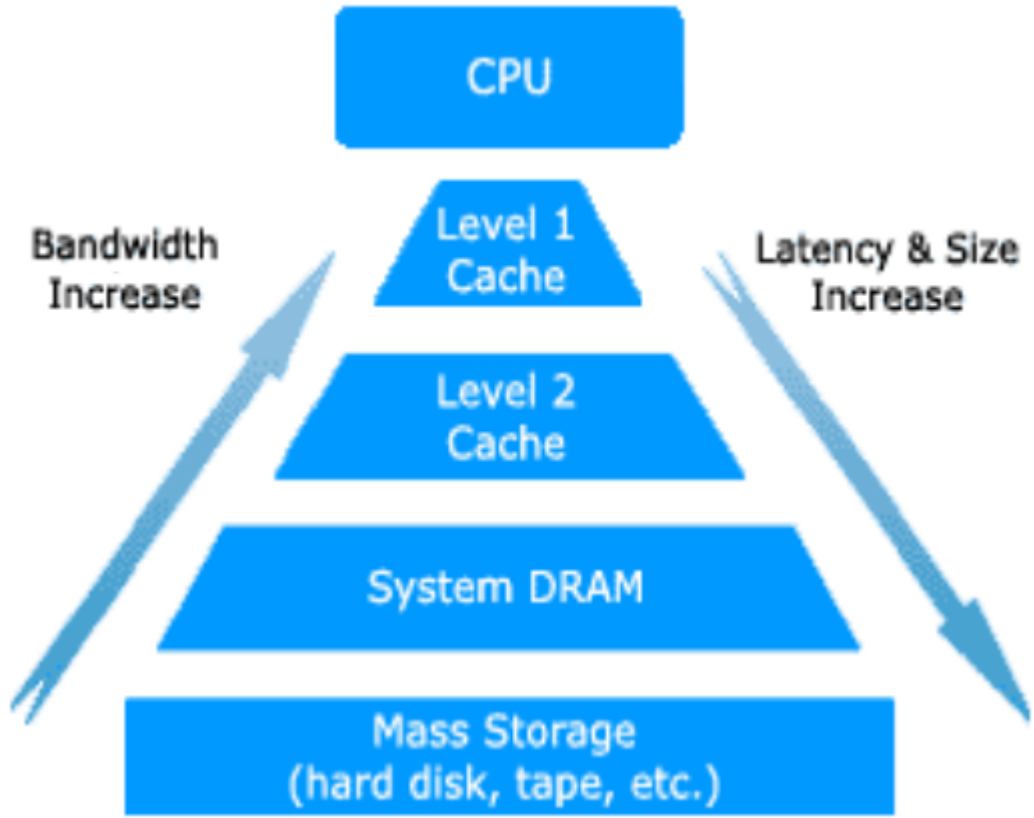
- Lars Hofhansl
- Architect at Salesforce
- HBase Committer
- Member of HBase PMC
- HBase blog: http://hadoop-hbase.blogspot.com
- Generalist.
  - have automated oil refineries using Fortran
  - hacked databases and appservers
  - implemented Javascript UI frameworks
  - mostly a backend plumber

- Aikido black belt

Part I - Performance

(some of the following slides are lifted/adapted from a talk my coworker Jamie Martin gave some time back)

Modern HW architecture



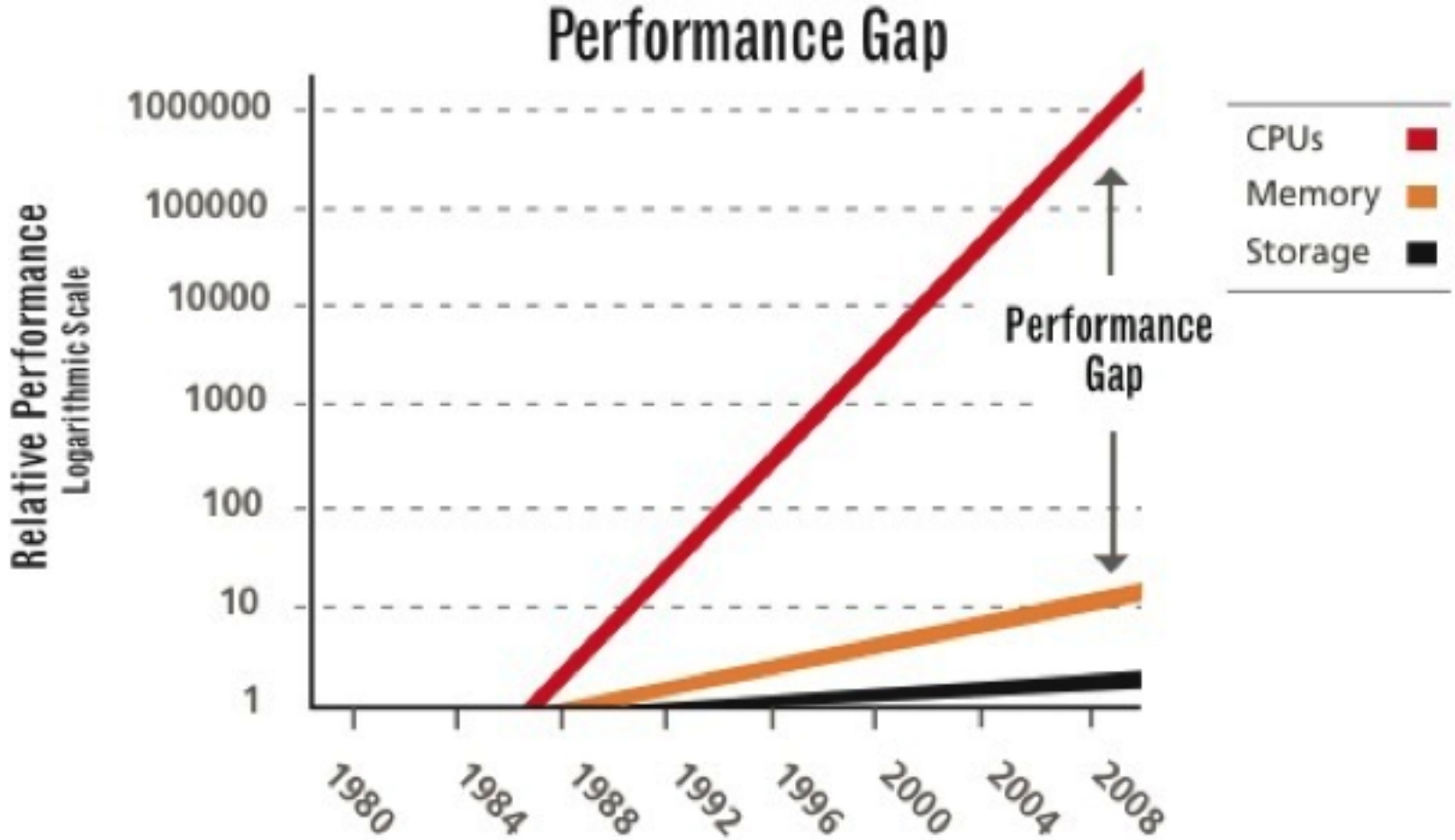
Jeff Dean's "Numbers Everyone Should Know"

- CPU ~3GHZ (.33ns cycle time)
- L1 cache reference .5ns
- L2 cache reference 7ns
- Main memory reference 100ns
- Read 1MB sequentially from memory 250,000ns
- Disk seek: 10,000,000ns
- Read 1MB sequentially from disk 20,000,000ns
- Read 1TB sequentially from disk: hours
- Read 1TB randomly from disk: 10s of days

Trends:

- CPU speed improves 60% per year, DRAM speed about 10%
- GAP grows 50% year over year
- Too much heat beyond 4 GHZ -> multiples cores, hw threads

Performance Gap: Memory and disks are getting further and further away from the CPU...



Implications:

- CPU is idle for **~300 cycles** during a memory fetch!
- RAM is the new disk, disk is the new tape
  (tape is the new horse-carriage delivery)
- Jim Gray: 100x faster cache at 90% hit rate makes effective perf improvement of 10x
- Each CPU core has its own L1/L2 cache.
  - cores can see each others changes out of order
  - Reordering limited via read/write barriers
- **Must** design with cache coherency/locality in mind
  - Avoid memory barriers
  - Avoid unnecessary copies in RAM
    (remember RAM is a slow shared resource now, that does not scale with the cores)

A memory barrier (or fence) ensures that all prior reads or writes are seen in the same order by all cores. (details are tricky, I am not a hardware guy)

Memory barriers in Java:

- `synchronized` - sets read and write barriers as needed (details depend on JVM, version, and settings)
- `volatile` - sets a read barrier before a read to a volatile, and write barrier after a write
- `final` - sets a write barrier after the assignment
- `AtomicInteger`, `AtomicLong`, etc - uses volatiles and hardware CAS instructions

TL;DR: Avoid these in hot code paths

But what about HBase?

HBASE-6603 - RegionMetricsStorage.incrNumericMetric is called too often

HBASE-6621 - Reduce calls to Bytes.toInt

HBASE-6711 - Avoid local results copy in StoreScanner

HBASE-7180 - RegionScannerImpl.next() is inefficient

HBASE-7279 - Avoid copying the rowkey in RegionScanner, StoreScanner, and ScanQueryMatcher

HBASE-7336 - HFileBlock.readAtOffset does not work well with multiple threads

HBASE-6852 - SchemaMetrics.updateOnCacheHit costs too much while full scanning a table with all of its fields (Cheng Hao)

(unscientific) Benchmarks

- Phoenix 1.1 improved 10% for aggregate queries, 25% for non-aggregate queries

- client scan performance:
  - 20m rows, ~50 bytes each, scan in batches of 10000 rows

	0.94.0	0.94.4
disk read	54s	50s (io bound, disk + net)
OS cache	51s	35s
block cache	35s	32s (limited to net to client)

  - 20m rows, ~50 bytes each, scan + filter everything (i.e. measure internal friction)

	0.94.0	0.94.4
disk read	31s	22s
OS cache	25s	17s
block cache	11s	6.3s

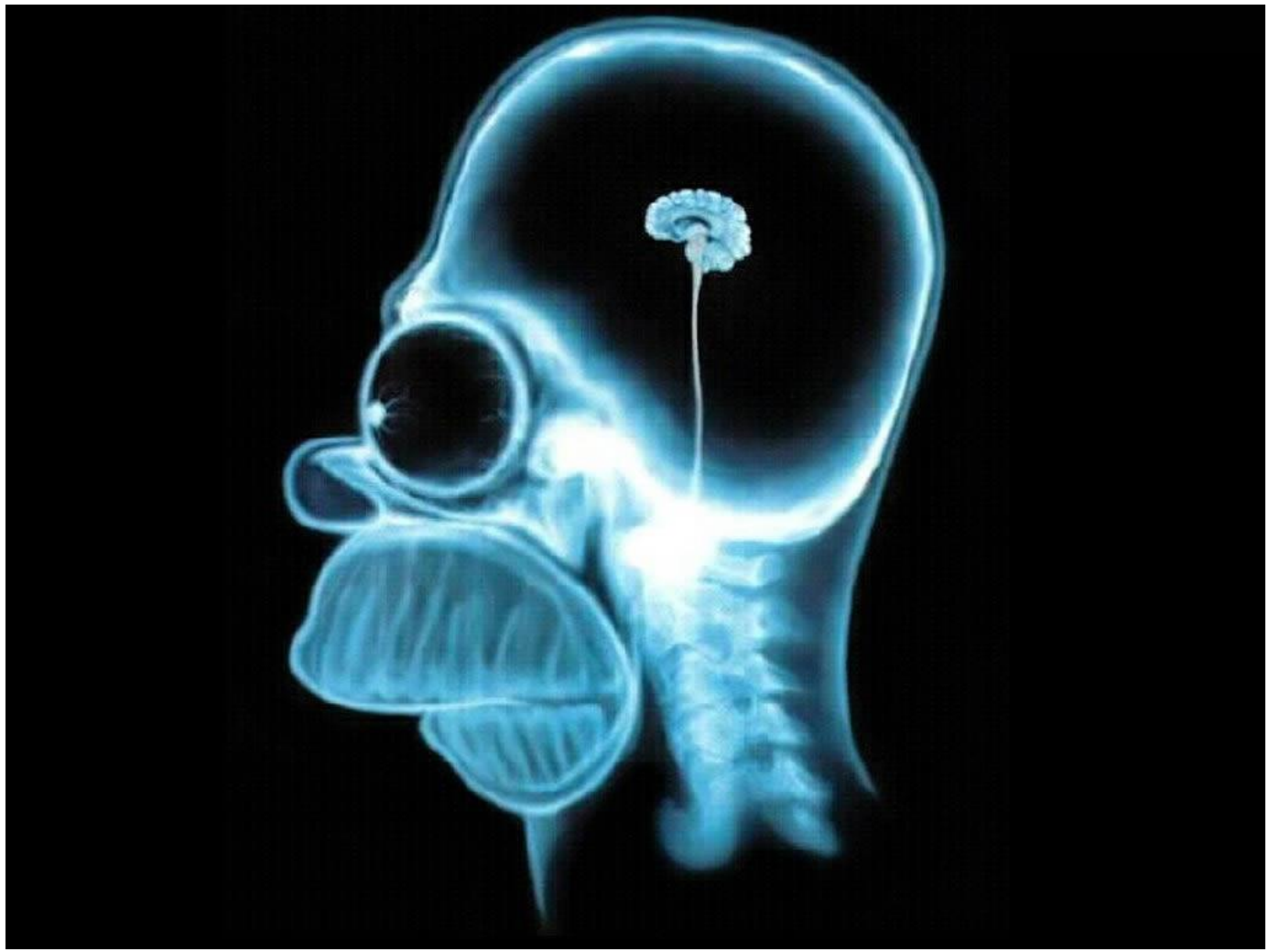
more (unscientific) Benchmarks

- Raw scan performance:
  - 20m rows, ~50 bytes each, scan + filter
  - 6 cores, 1.5ghz, 12 HTs

	0.94.0	0.94.5
1 thread	8.3s	7.1s
2 threads	10.6s	8.3s
3 threads	11.5s	9.1s
4 threads	12.5s	9.5s
5 threads	15.5s	9.8s
6 threads	18.1s	9.8s
12 threads	25s	17s

Note: with 6 threads we're scanning over 12m rows/s on a single machine in 0.94.5.
Doesn't scale far beyond number of cores.

Part II - Lesser known HBase features



(Or is it just a crayon up the nose?)

1. KEEP_DELETED_CELLS
2. MIN_VERSIONS
3. Atomic Row Mutations
4. MultiRowMutationEndpoint
5. BulkDeleteEndpoint
6. Filters and essential column families

KEEP_DELETED_CELLS

- HBase allows flashback queries
  - Ability to get the state of your data as of a time
  - Scan.setTimeRange, Scan.setTimeStamp
- This does not work with deletes
  - Put(X) at TS
  - Put(X') at TS+1
  - Scan at TS: X
  - Delete(X) at TS+2
  - Scan at TS or TS+1: nil
- KEEP_DELETED_CELLS keeps deleted cells until they naturally expire (TTL or number of versions)
  - Scan at TS: X
  - Scan at TS+1: X'

MIN_VERSIONS

- HBase support TTL (compliment of your host Andy Purtell)
  - data in column family is assigned a maximum life time, after that it is unconditionally deleted.
- What if you wanted to expire your data after a while, but keep at least the last (N) version(s)?
  - For example you want to keep the 24h of data, but keep at least one version if no change was made in 24h.
- Enter MIN_VERSIONS.
  - Works together with TTL.
  - Defines the number of versions to retain even when their TTL has expired

Atomic Row Mutations

- Atomically apply **multiple** operation to a row:

```
HTable t = ...;
byte[] row = ...;
RowMutations rm = new RowMutations(row);
Put p = new Put(row);
p.add(...)
Delete d = new Delete(row);
p.delete{Column|Columns|Family}(...);
arm.add(p);
arm.add(d);
t.mutateRow(rm);
```

- RowMutations implements the Row interface:

```
RowMutation arm1 = new RowMutation(row1);
RowMutation arm2 = new RowMutation(row2);
...
List<Row> rows = ...;
rows.add(arm1);
rows.add(arm2);
t.batch(rows);
```

Multi Row Transactions

- **HBase supports multi row transaction if rows are colocated in the same regions**
- **Colocation is controlled by a SplitPolicy**
- **Transactions are implemented in** MultiRowMutationEndpoint

Create a table that uses KeyPrefixRegionSplitPolicy:

```
HTableDescriptor myHtd = new HTableDescriptor("myTable"); myHtd.
setValue(HTableDescriptor.SPLIT_POLICY,
                KeyPrefixRegionSplitPolicy.class.getName());
// set the prefix to 3 in this example
myHtd.setValue(KeyPrefixRegionSplitPolicy.PREFIX_LENGTH_KEY,
              String.valueOf(3));
HColumnDescriptor hcd = new HColumnDescriptor(...);
myHtd.addFamily(hcd);
HBaseAdmin admin = ...;
admin.createTable(myHtd);
```

Execute an atomic multirow transaction:

```
List<Mutation> ms = new ArrayList<Mutation>();
Put p = new Put(Bytes.toBytes("xxxabc"));
...
ms.add(p);
Put p1 = new Put(Bytes.toBytes("xxx123"));
...
ms.add(p1);
Delete d = new Delete(Bytes.toBytes("xxxzzz"));
...
ms.add(d);
// get a proxy for MultiRowMutationEndpoint
MultiRowMutationProtocol mr = t.coprocessorProxy(
   MultiRowMutationProtocol.class,
   Bytes.toBytes("xxxabc"));
// perform the atomic operation
mr.mutateRows(ms);
```

Filter with essential families

- Filters in HBase are powerful
  - Skip scans via hinting
- But need to load entire row to evaluate filter
- Some filter don't need the entire row
- New feature in 0.94.5: FilterBase.isFamilyEssential(byte[] cfname)
  - Filter will only load essential column families.
  - In a 2nd pass all column families are loaded as needed (i.e. if the included the rest of the row)
- Currently only implemented by SingleColumnValueFilter

Bulk Deletes

- Problem: Delete all column families, or columns, or column versions that are matched by a Scan.
- BulkDelete coprocessor endpoint shipped with HBase addresses that problems:

```
Scan scan = new Scan();
// set scan properties(rowkey range, filters, timerange, etc).
HTable ht = ...;
Batch.Call<BulkDeleteProtocol, BulkDeleteResponse> callable =
      new Batch.Call<...>() {
  public BulkDeleteResponse call(BulkDeleteProtocol instance) throws IOException {
      return instance.deleteRows(scan, BulkDeleteProtocol.DeleteType,
                                          timestamp, rowBatchSize);
  }
};
Map<byte[], BulkDeleteResponse> result = ht.coprocessorExec(BulkDeleteProtocol.class,
      scan.getStartRow(), scan.getStopRow(), callable);
 for (BulkDeleteResponse response : result.values()) {
      response.getRowsDeleted(); // return # of deleted rows
 }
```

Future

(I am making this up, this is not an official statement in any way)

More possible performance improvements

- True KeyValue interface to allow KeyValues based on scattered byte[]. This is very helpful for prefix encoding.
- New HFile layout which separates key, column families, columns, and values. Can then efficiently scan along along a column without touching the key, column, etc.
- Possibly some more low hanging fruit
- RPC improvements

Features

- Key encoding library (Nick Dimiduk is working on HBASE-7692)
- System tables and table open dependencies
- Statistics equal depth and/or equal width histograms collected during compactions
- Basic building blocks for 2ndary indexes