

# Covering Indexes: Orders-of-Magnitude Improvements

Bradley C. Kuszmaul  
Chief Architect

**Tokutek**<sup>TM</sup>

Percona Performance Conference 2009

# A Performance Example

A fact table drawn from iiBench, except no indexes. Like TPCH, the **int** values are essentially random.

```
Create Table: CREATE TABLE 'facts' (  
  'xid' int(11) NOT NULL AUTO_INCREMENT,  
  'dateandtime' datetime DEFAULT NULL,  
  'cashregisterid' int(11) NOT NULL,  
  'cust_id' int(11) NOT NULL,  
  'prod_id' int(11) NOT NULL,  
  'price' float NOT NULL,  
  PRIMARY KEY ('xid'),  
) ENGINE=TOKUDB
```

*Populated with 1 billion rows.*

# The Query

A simple query:

```
mysql> select prod_id from facts
        where cust_id = 50000;
```

prod_id
525
654
704
⋮
984
276
576

**10014 rows in set (11 min 26.26 sec)**

*Implemented via table scan.  
14.6 rows/s.*

# An Index Doubles Speed

```
mysql> alter table add index facts
      cust_idx(cust_id);
mysql> select prod_id from facts
      where cust_id = 50000;
```

prod_id
525
654
704
⋮
984
276
576

**10014 rows in set (5 min 48.94 sec)**

*Looks at 0.001% of the data and get 2x speedup.  
29 rows/s.*

# Covering Index

A *covering index* is an index in which all the necessary columns are part of the key.

```
mysql> alter table add index facts  
      cust_prod_idx (cust_id, prod_id);
```

*Even if we have a key that happens to be unique, we can throw in some more keys (which doesn't change the index order) to make it a covering index.*

# Covering Index Is 1300x Faster

```
mysql> alter table add index facts  
        cust_prod_idx (cust_id, prod_id);  
mysql> select prod_id from facts  
        where cust_id = 50000;
```

prod_id
0
0
⋮
999
999

10014 rows in set (0.26 sec)

*Note different row order.*

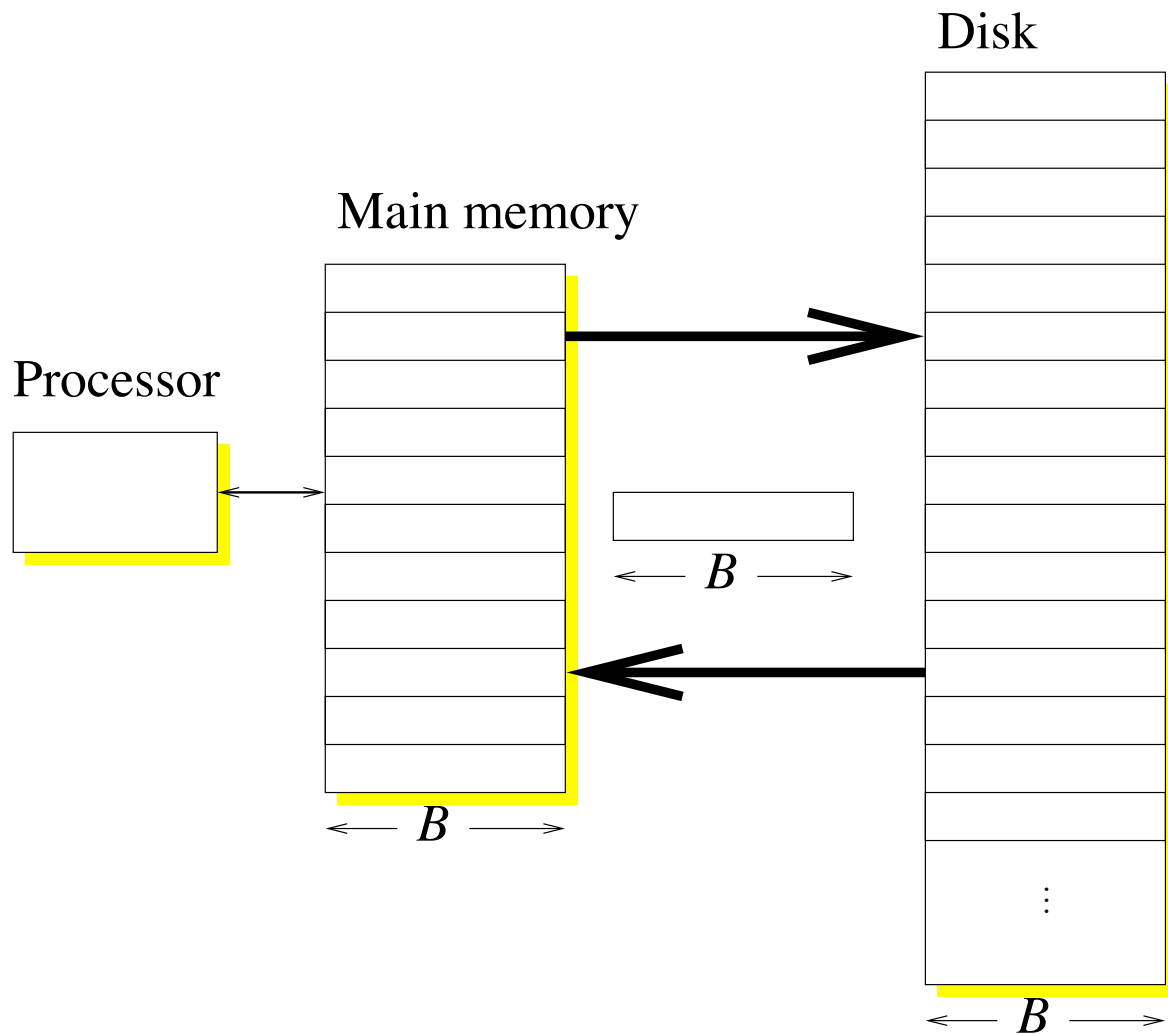
# Outline

For a database that doesn't fit in main memory, how can we predict performance?

- The Disk Access Model (DAM).
- Analysis using the DAM.
- Which indexes should we maintain?
- Another brief example: TPC-H.
- Does SSD help?

*In this talk, I'll describe a theoretical model for predicting performance, and show how to use it.*

# The Disk-Access Model



A theoretical model for understanding performance of data structures on disk. Memory is organized in blocks of size  $B$ .

Blocks are transferred between memory and disk. Count only the number of block transfers.

*Model can predict the performance of a query plan.*

# Analysis for No Index

	xid	cust_id	prod_id	
$10^9$ rows	1	42	501	Block 0 (size $B$ )
		⋮		
	6044	50000	525	
		⋮		Block 1
	20480	50000	654	
		⋮		
	44921	50000	704	Block $10^9/B - 2$
		⋮		
	999703368	50000	984	
		⋮		Block $10^9/B - 1$
999850921	50000	276		
	⋮			
999923451	50000	576	Block $10^9/B$	
	⋮			

*Table scan requires  $O(10^9/B)$  transfers.*

# Analysis for Index

**cust\_idx**

cust_id	xid
42	1
⋮	
50000	6044
50000	20480
50000	44921
⋮	
50000	999703368
50000	999850921
50000	999923451
⋮	

**facts**

xid	cust_id	prod_id
1	42	501
⋮		
6044	50000	525
⋮		
20480	50000	654
⋮		
44921	50000	704
⋮		
999703368	50000	984
⋮		
999850921	50000	276
⋮		
999923451	50000	576
⋮		

Fetch only **10014** rows,  
but mostly in different  
blocks, so  $O(10014)$   
memory transfers.

# Analysis With Covering Index

**cust\_prod\_idx**

cust_id	prod_id	xid
42	501	1
50000	276	999850921
50000	525	6044
50000	576	999923451
50000	654	20480
50000	704	44921
50000	984	999703368

**facts**

xid	cust_id	prod_id
1	42	501
:	:	:
6044	50000	525
:	:	:
20480	50000	654
:	:	:
44921	50000	704
:	:	:
999703368	50000	984
:	:	:
999850921	50000	276
:	:	:
999923451	50000	576
:	:	:

Answer **directly out of the index**:  $O(10014/B)$  transfers.  
(The prod\_ids are sorted for each customer.)

# Which Indexes?

**Problem:** MySQL only allows 16 columns in an index (32 in TokuDB).

Covering indexes speed up queries.

But which columns should we throw in?

**Solution:** We defined *clustering indexes*.

A clustering index is an index which includes all the columns in the index.

```
mysql> alter table facts add clustering index  
      cust_cluster_idx (customerid);
```

Materializes a table, sorted in a different order, clustered on the index.

*A clustering index is a covering index for all queries.*

# TPC-H Q17

We've been trying to figure out how to make TPC-H-like queries run faster, so we picked Q17, which is one of the slowest in MySQL.

Results: With a clustering index on **(L\_PARTNUM, L\_QTY)**:

Scale	Standard	Clustering
SF10 (10GB)	> 3600s	101s (> 36x speedup)
SF100 (100GB)	680533s	773s (770x speedup)

I'll write more about this in blog [tokuvview.com](http://tokuvview.com).

# Indexes Are Expensive (or Are They?)

The downside of maintaining indexes is that insertions are more expensive.

Fractal Tree indexes speed up insertions by orders of magnitude, however.

For  $B = 4096$  rows and  $N = 10^9$  rows, the number of memory transfers per operation is

	B-Trees	Fractal Trees
Point Query	$O\left(\frac{\log N}{\log B}\right) \geq 1$	$O(\log_B N) \geq 1$
Range Query	$O(S/B)$	$O(S/B)$
Insertion	$O\left(\frac{\log N}{\log B}\right) \geq 1$	$O\left(\frac{\log N}{B}\right) = 0.007$

*TokuDB, the Tokutek storage engine, implements Fractal Tree indexes.*

# Other Materialization Ideas

Better tools are needed to help maintain other interesting materializations for MySQL. For example:

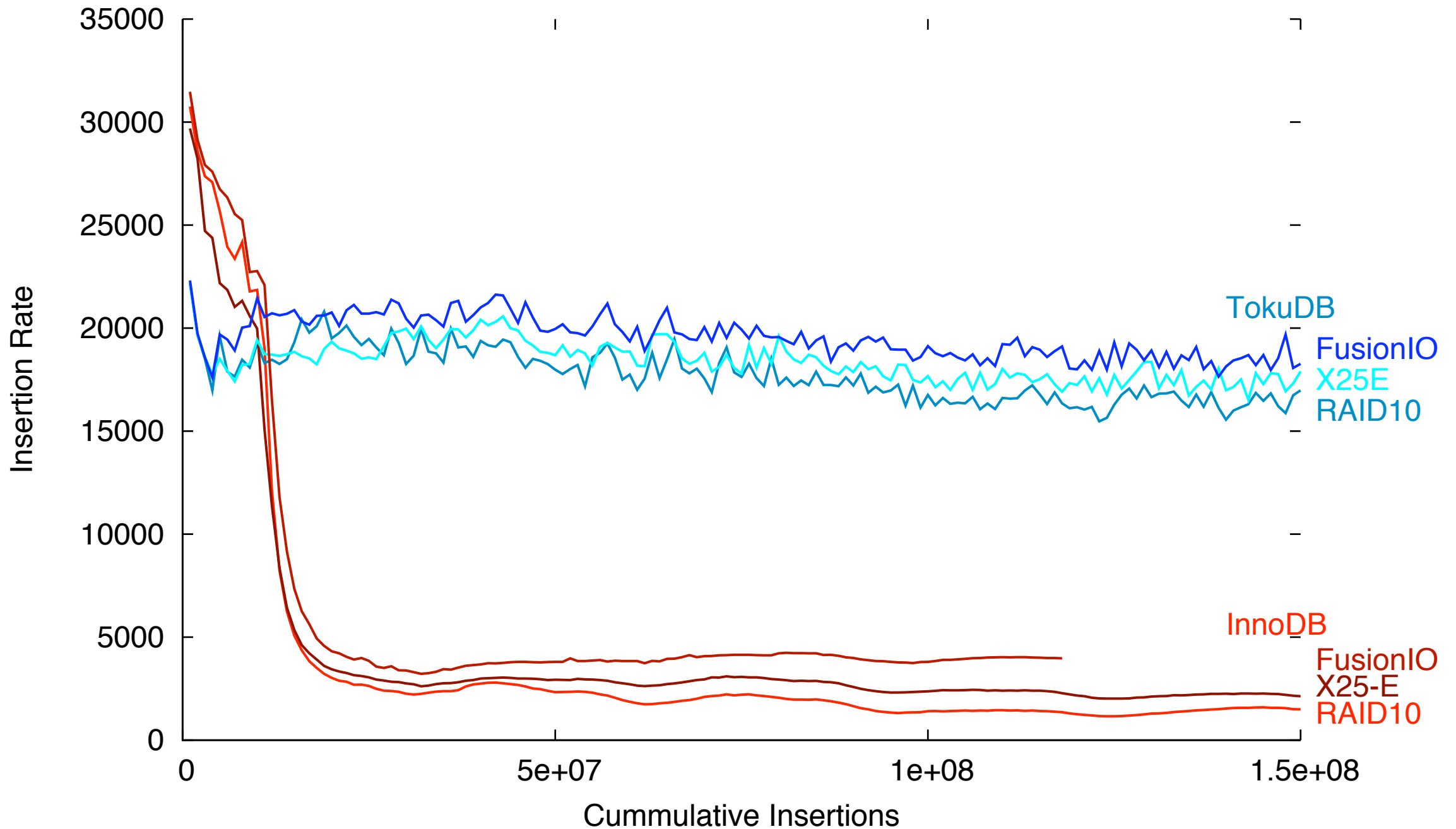
- Denormalization: prejoining some columns.
- Multidimensional indexing: often **where** clauses look like range queries on multiple columns.

**$38 \leq a$  and  $a \leq 42$  and  $90 \leq b$  and  $b \leq 99$**

Partions can provide a painful substitute for multidimensional indexing.

*Fractal Tree indexes can help solve these problems.*

# iiBench on RAID10 and SSD



Percona measured TokuDB and InnoDB on iiBench on RAID 10 disks, Intel X25-E 32GB SSD, and FusionIO 160GB SSD.

*These SSDs provide surprisingly little performance.*