Mastering the art of indexing

Yoshinori Matsunobu

Lead of MySQL Professional Services APAC

Sun Microsystems

Yoshinori.Matsunobu@sun.com

Table of contents

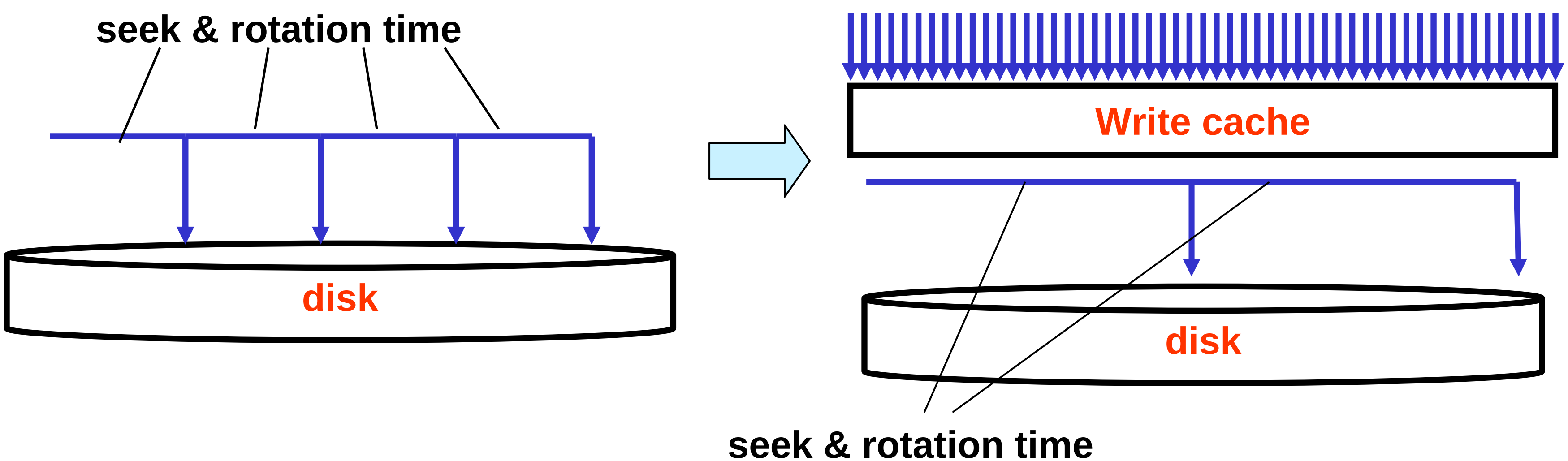
- Speeding up Selects
  - B+TREE index structure
  - Index range scan, MySQL 6.0 MRR
  - Covering Index (Index-only read)
  - Multi-column index , index-merge

- Insert performance
  - Understanding what happens when doing insert
  - “Insert buffering” in InnoDB

- Benchmarks
  - SSD/HDD
  - InnoDB/MyISAM
  - Changing RAM size
  - Using MySQL 5.1 Partitioning
  - Changing Linux I/O scheduler settings

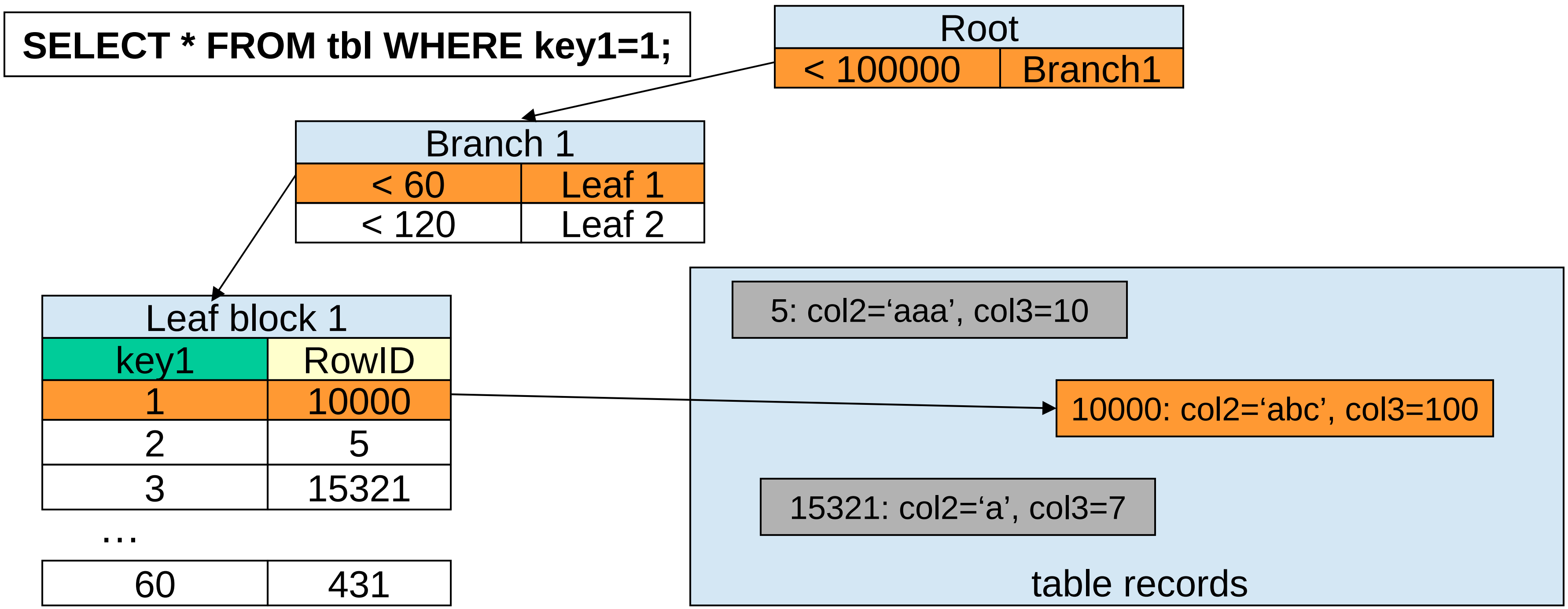
Important performance indicator: IOPS

- Number of (random) disk i/o operations per second
- Regular SAS HDD : 200 iops per drive (disk seek & rotation is heavy)
- Intel SSD (X25-E): 2,000+ (writes) / 5,000+ (reads) per drive
  - Currently highly depending on SSDs and device drivers
- Best Practice: Writes can be boosted by using BBWC
  (Battery Backed up Write Cache), especially for REDO Logs



Speeding up Selects

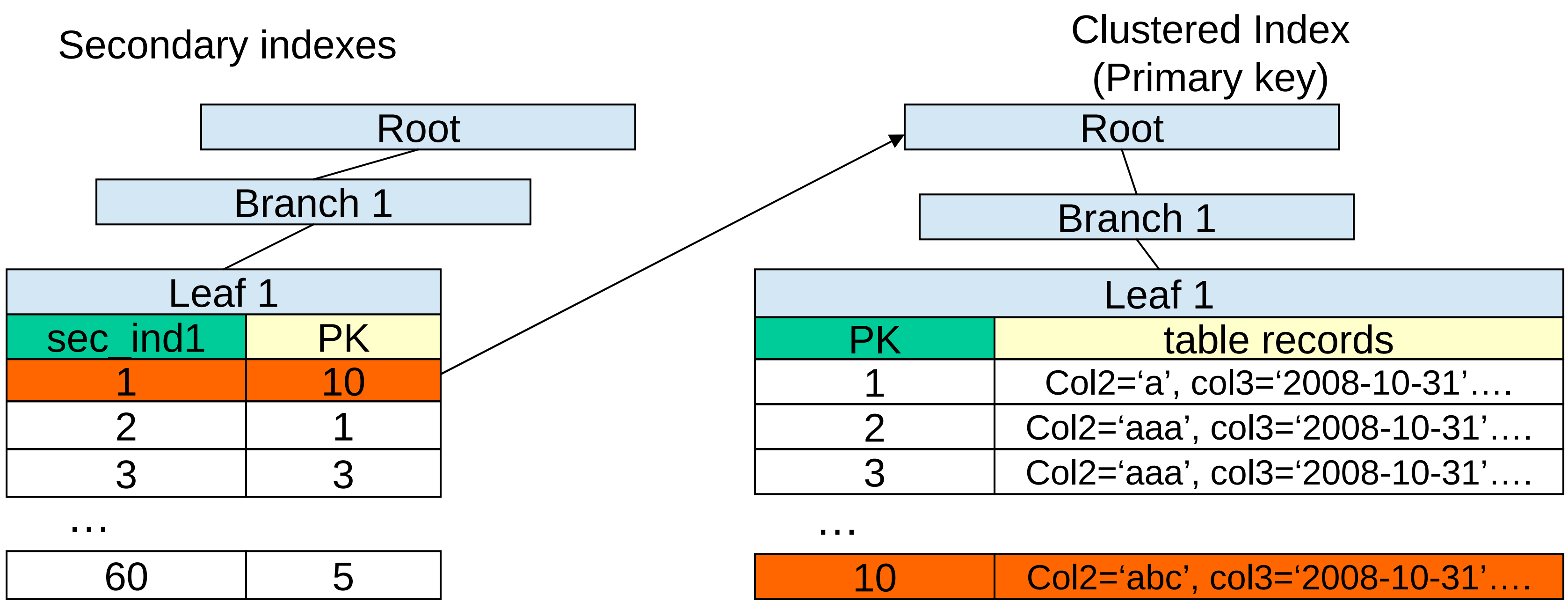
B+Tree index structure



- Index scan is done by accessing "Root" -> "Branch" -> "Leaf" -> table records
- Index entries are stored in leaf blocks, sorted by key order
- Each leaf block usually has many entries
- Index entries are mapped to RowID.
  - -RowID = Record Pointer (MyISAM), Primary Key Value (InnoDB)
- Root and Branch blocks are cached almost all times
- On large tables, some(many) leaf blocks and table records are not cached

InnoDB: Clustered Index (index-organized table)

SELECT * FROM tbl WHERE secondary_index1=1;

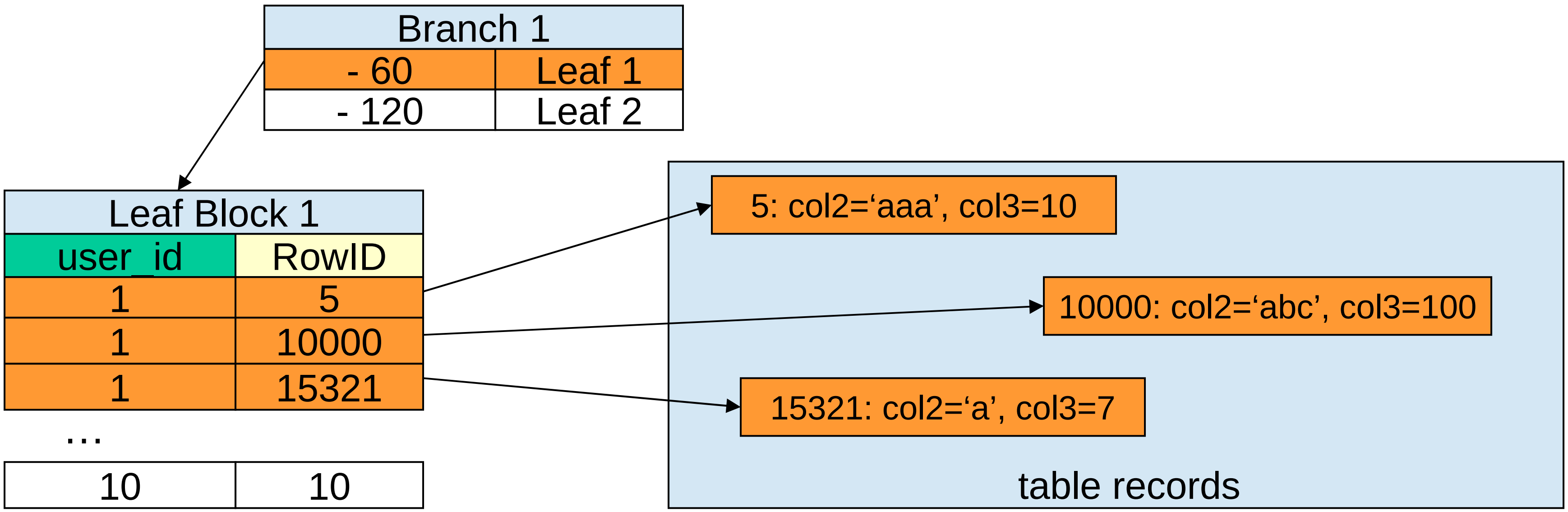


- On Secondary indexes, entries are mapped to PK values
  - Two-step index lookups are required for SELECT BY SECONDARY KEY

- Primary key lookup is very fast because it is single-step
  - Only one random read is required for SELECT BY PRIMARY KEY

Non-unique index scan

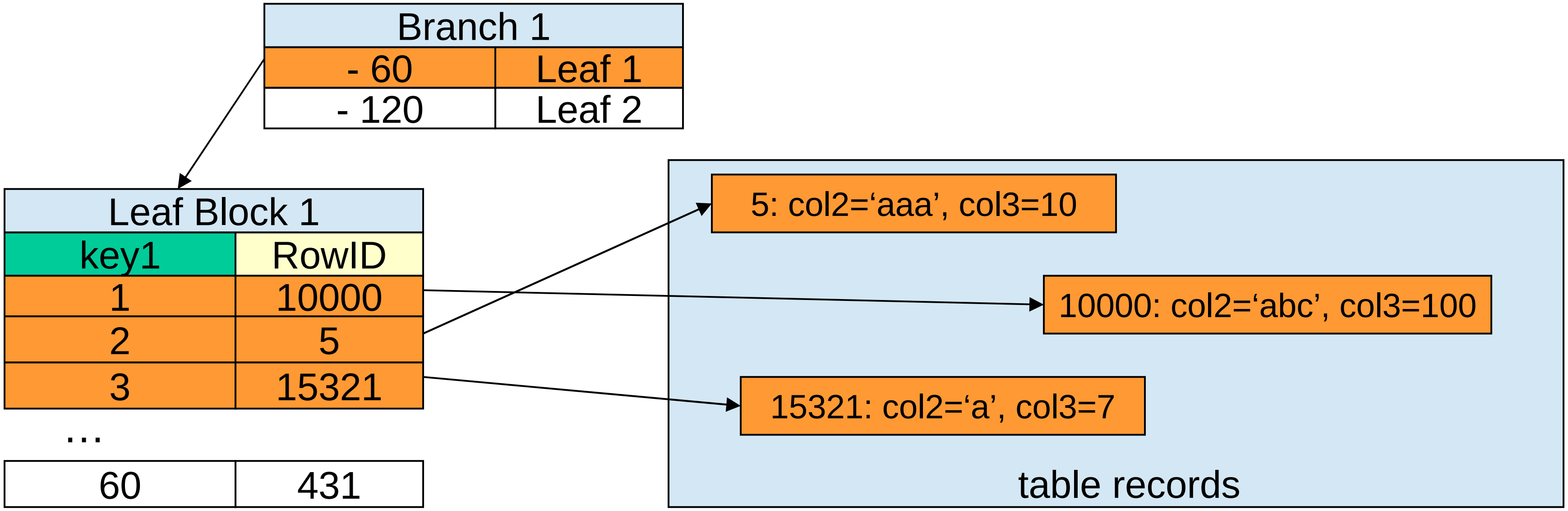
SELECT * FROM message_table WHERE user_id =1;



- When three index entries meet conditions, three random reads might happen to get table records
- Only one random read for the leaf block because all entries are stored in the same block
- 1 time disk i/o for index leaf, 3 times for table records (1 + N random reads)

Range scan

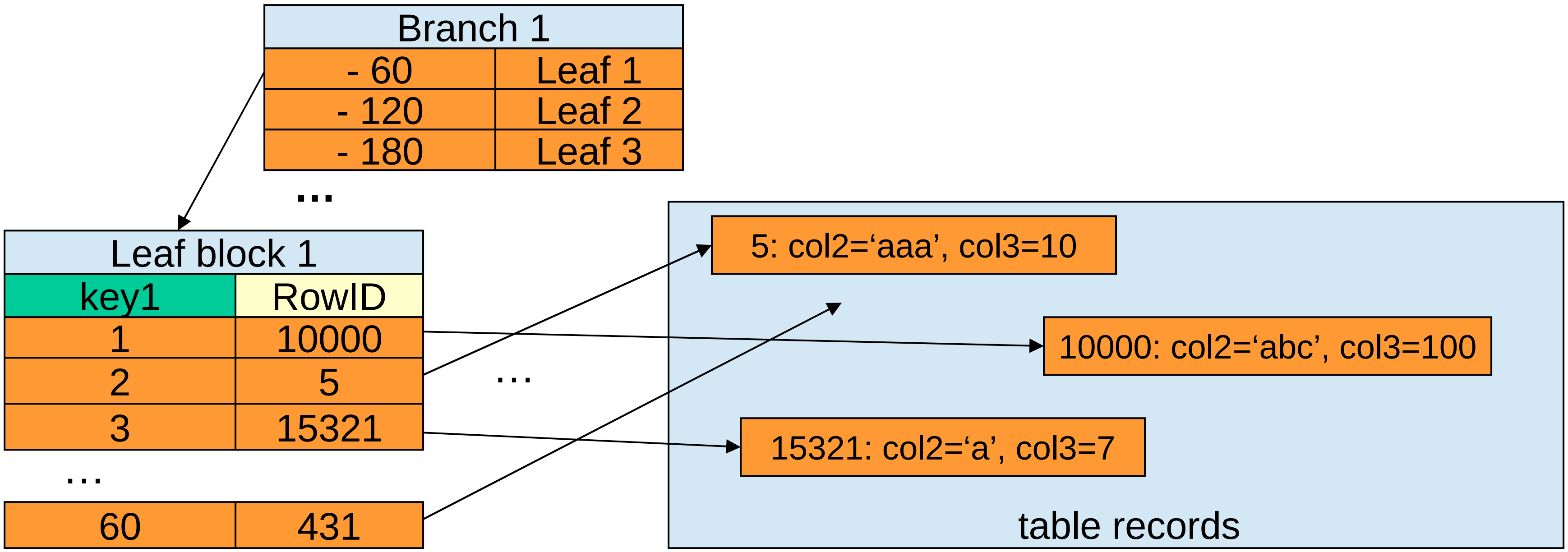
SELECT * FROM tbl WHERE key1 BETWEEN 1 AND 3;



- key 1 .. 3 are stored in the same leaf block -> only one disk i/o to get index entries
- RowIDs skip, not sequentially stored
  -> Single disk read can not get all records, 3 random disk reads might happen
- 1 time disk i/o for index leaf, 3 times for table records (1 + N random reads)

Bad index scan

SELECT * FROM tbl WHERE key1 < 2000000



- When doing index scan, index leaf blocks can be fetched by *sequential* reads, but *random* reads for table records are required (very expensive)
- Normally MySQL doesn't choose this execution plan, but choose full table scan (Sometimes not, control the plan by IGNORE INDEX)
- Using SSD boosts random reads

Full table scan

SELECT * FROM tbl WHERE key1 < 2000000

Branch 1	
- 60	Leaf 1
- 120	Leaf 2
- 120	Leaf 3

Leaf Block 1	
key1	RowID
1	10000
2	5
3	15321
…	
60	431

…

Full table scan

5: col2='aaa', col3=10

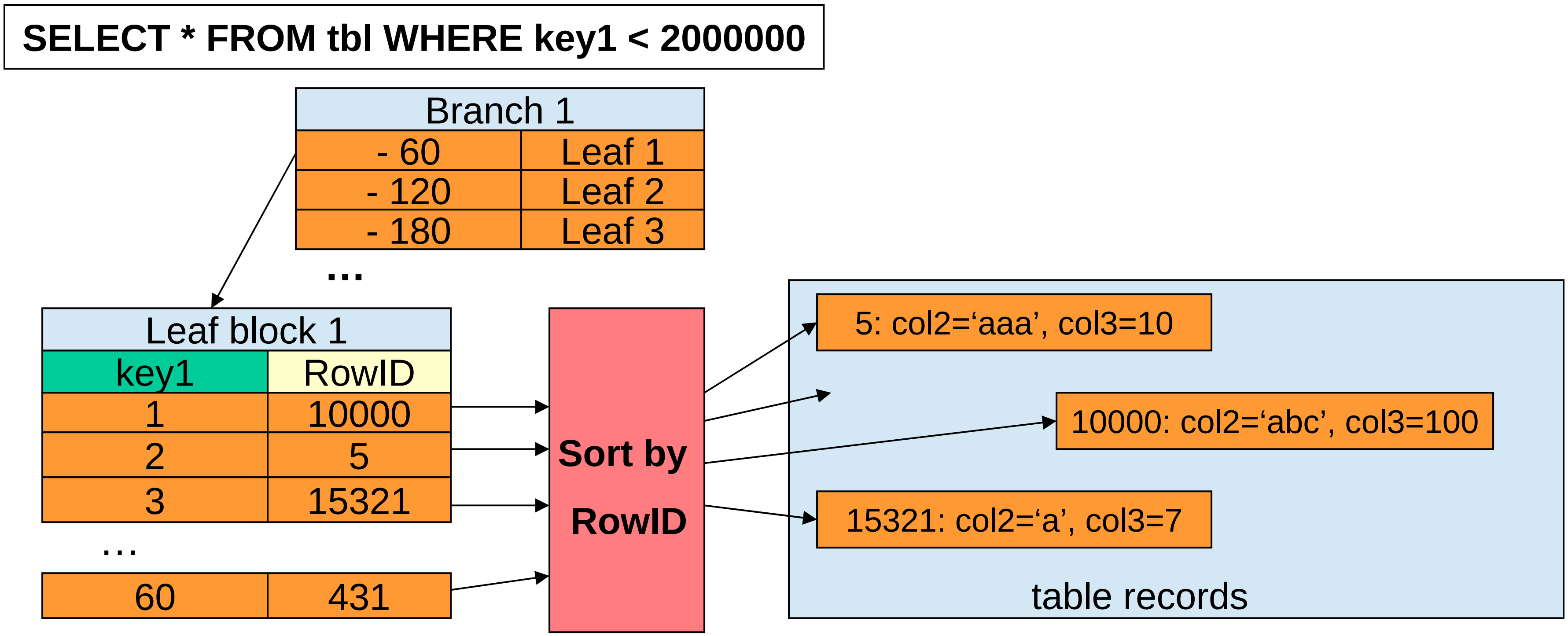
10000: col2='abc', col3=100

15321: col2='a', col3=7

table records

- Full table scan does sequential reads
- Reading more data than bad index scan, but faster because:
  - Number of i/o operations are much smaller than bad index scan because:
    - Single block has a number of records
    - InnoDB has read-ahead feature. Reading an extent (64 contiguous blocks) at one time
    - MyISAM: Reading read_buffer_size (128KB by default) at one time

Multi-Range Read (MRR: MySQL6.0 feature)



- Random read overheads (especially disk seeks) can be decreased
- Some records fitting in the same block can be read at once

Benchmarks : Full table scan vs Range scan

- Selecting 2 million records from a table containing 100 million records
  - Selecting 2% of data
  - Suppose running a daily/weekly batch jobs…

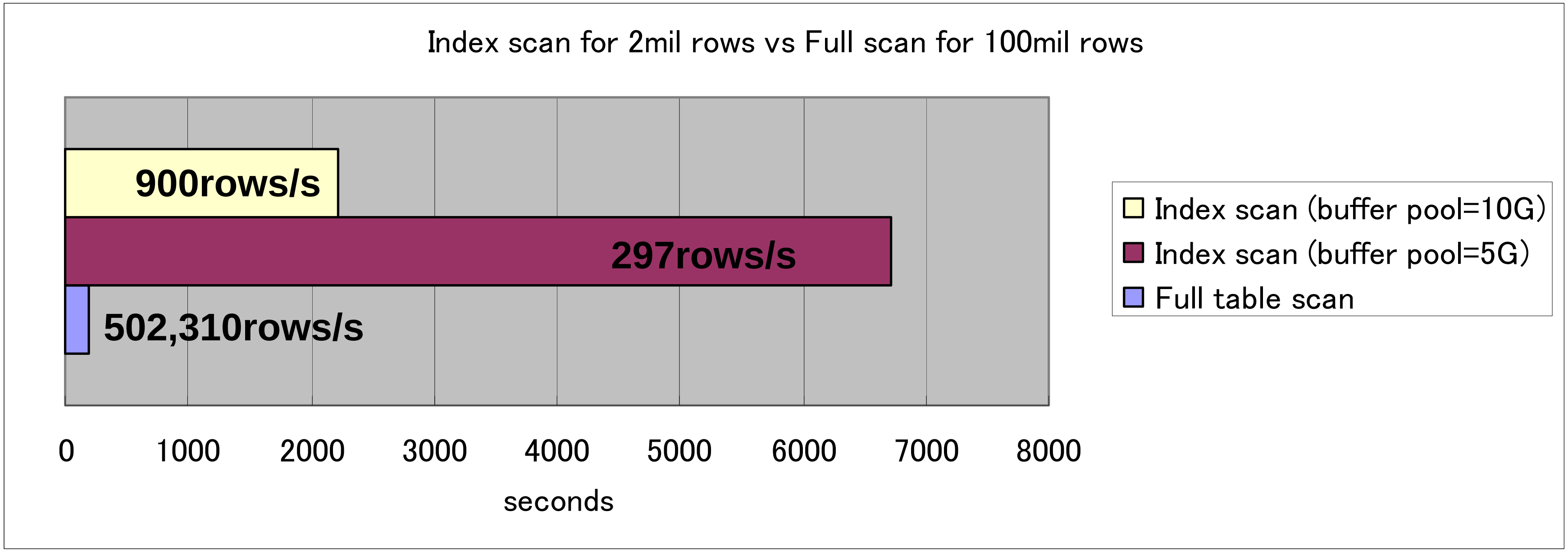
- Query: SELECT * FROM tbl WHERE seconday_key < 2000000
  - Secondary key was stored by rand()*100,000,000 (random order)

- Using InnoDB(built-in), 5.1.33, RHEL5.3 (2.6.18-128)

- 4GB index size, 13GB data (non-indexed) size
- Innodb_buffer_pool_size = 5GB, using O_DIRECT
- Not all records fit in buffer pool

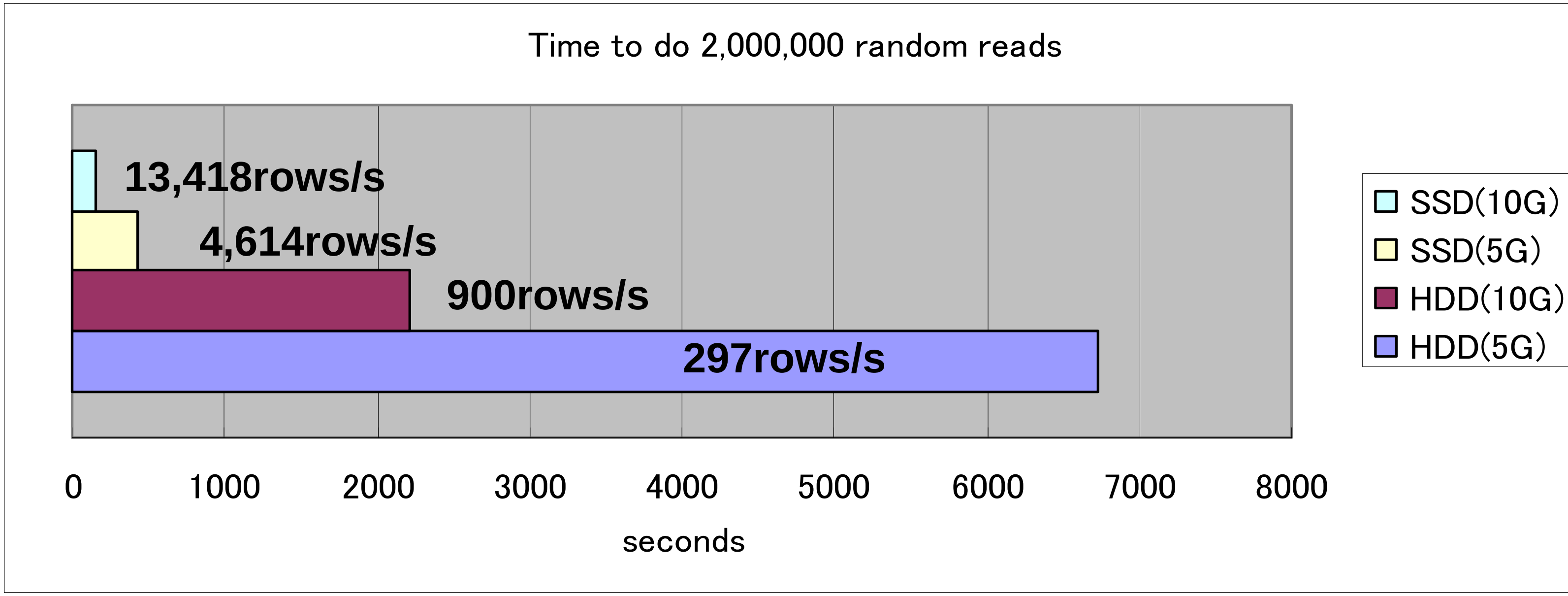
- Benchmarking Points
  - Full table scan (sequential access) vs Index scan (random access)
    - Controlling optimizer plan by FORCE/IGNORE INDEX
  - HDD vs SSD
  - Buffer pool size (tested with 5GB/10GB)

Benchmarks (1) : Full table scan vs Index scan (HDD)



- HDD: SAS 15000RPM, 2 disks, RAID1
- Index scan is 10-30+ times slower even though accessing just 2% records (highly depending of memory hit ratio)
- MySQL unfortunately decided index scan in my case (check EXPLAIN, then use IGNORE INDEX)

Benchmarks (2) : SSD vs HDD, index scan



- HDD: SAS 15000RPM, 2 disks, RAID1
- SSD: SATA Intel X25-E, no RAID
- SSD was 15 times faster than HDD
- Increasing RAM improves performance because hit ratio is improved (5G -> 10G, 3 times faster on both SSD and HDD)

OS statistics

HDD, range scan

#iostat -xm 1

	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	svctm	%util
sdb	0.00	0.00	243.00	0.00	4.11	0.00	34.63	1.23	5.05	4.03	97.90

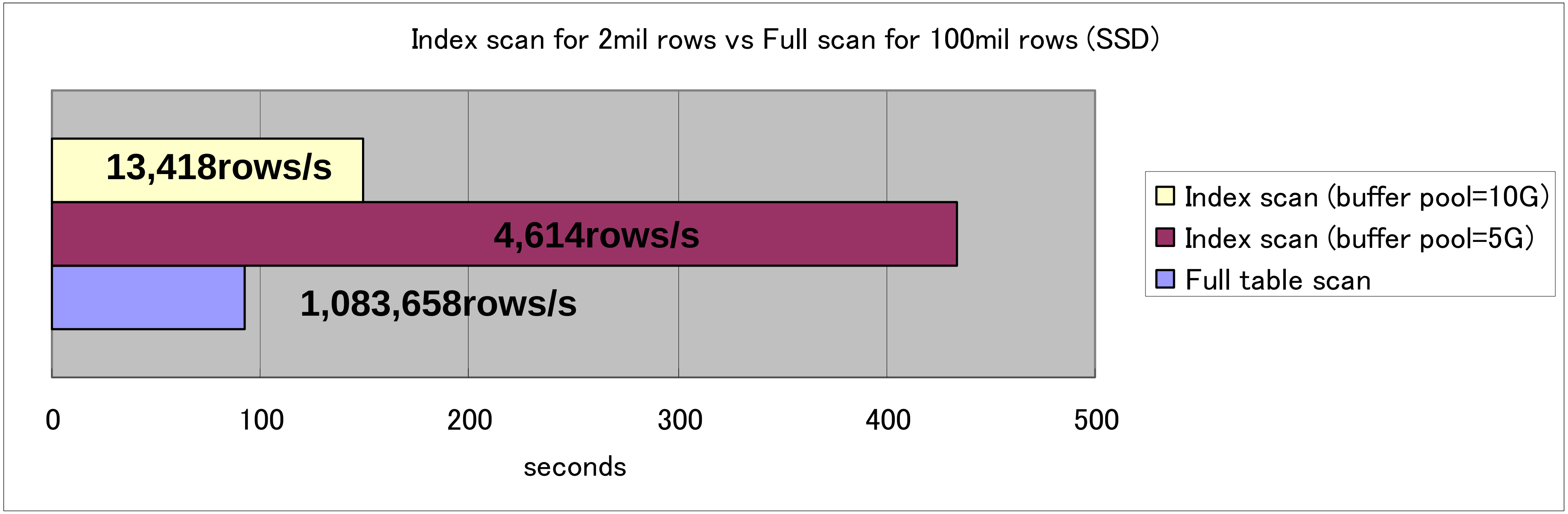
SSD, range scan

iostat -xm 1

	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	svctm	%util
sdc	24.00	0.00	2972.00	0.00	53.34	0.00	36.76	0.72	0.24	0.22	66.70

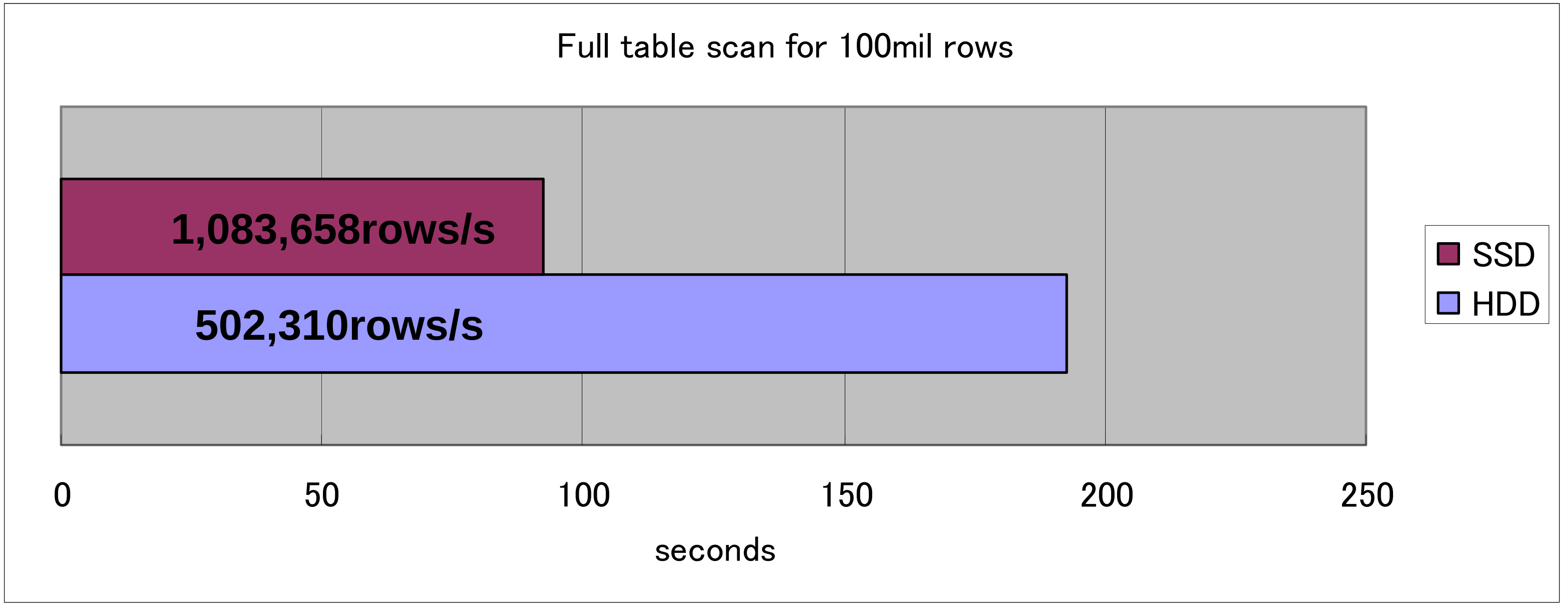
4.11MB / 243.00 ~= 53.34MB / 2972.00 ~= 16KB (InnoDB block size)

Benchmarks (3) : Full table scan vs Index scan (SSD)



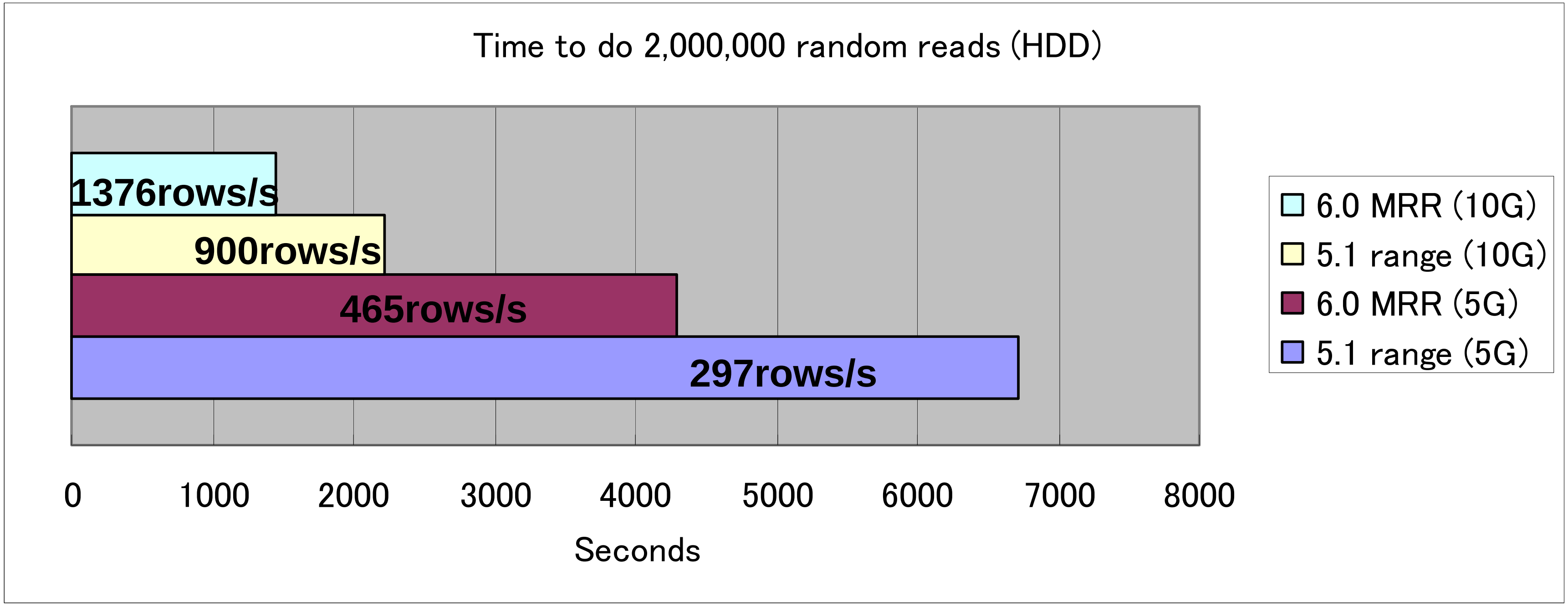
- SSD: SATA Intel X25-E
- Index scan is 1.5-5 times slower when accessing 2% of tables (still highly depending of memory hit ratio)
- Not so much slower than using HDD
- This tells that whether MySQL should choose full table scan or index scan depends on storage (HDD/SSD) and buffer pool
  - FORCE/IGNORE INDEX is helpful

Benchmarks (4) : SSD vs HDD, full scan



- Single SSD was two times faster than two HDDs

Benchmarks(5) : Using MySQL6.0 MRR

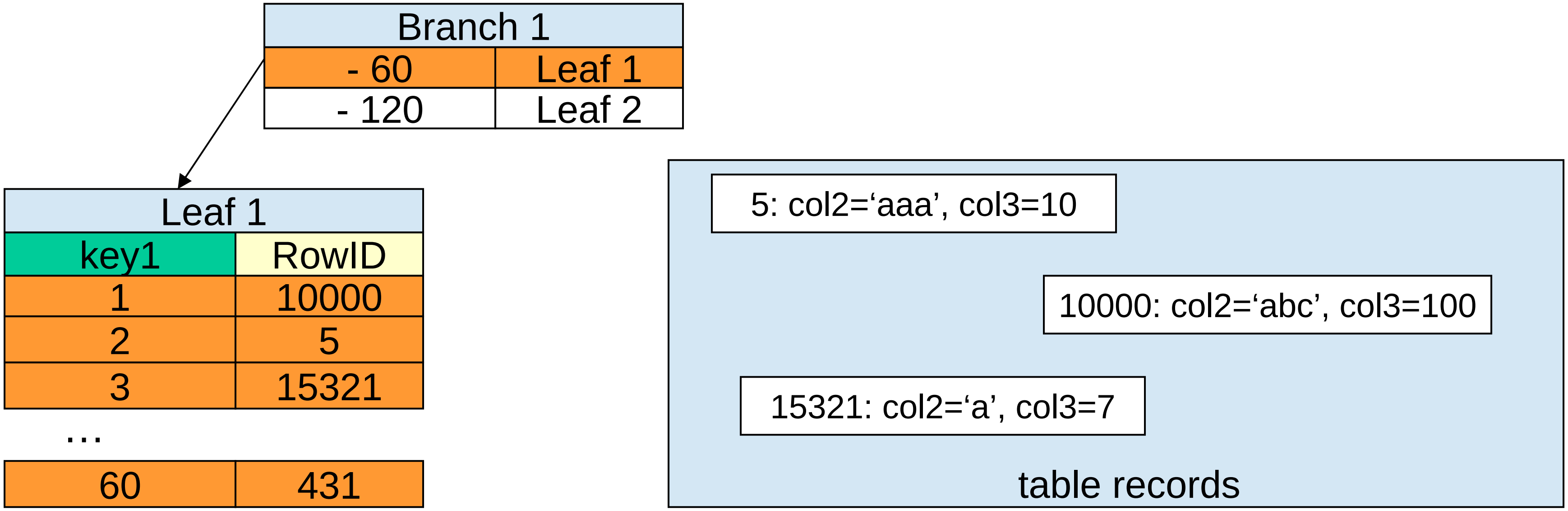


- 1.5 times faster, nice effect
- No performance difference was found on SSD

Covering index,
Multi column index, index merge

Covering Index (Index-only read)

SELECT key1 FROM tbl WHERE key1 BETWEEN 1 AND 60;



- Some types of queries can be completed by reading only index, not reading table records
- Very efficient for wide range queries because no random read happens
- All columns in the SQL statement (SELECT/WHERE/etc) must be contained within single index

Covering Index

```
> explain select count(ind) from t
           id: 1
  select_type: SIMPLE
        table: t
         type: index
possible_keys: NULL
          key: ind
      key_len: 5
          ref: NULL
         rows: 100000181
        Extra: Using index

mysql> select count(ind) from d;
+----------------+
|   count(ind)   |
+----------------+
|      100000000 |
+----------------+
1 row in set (15.98 sec)
```

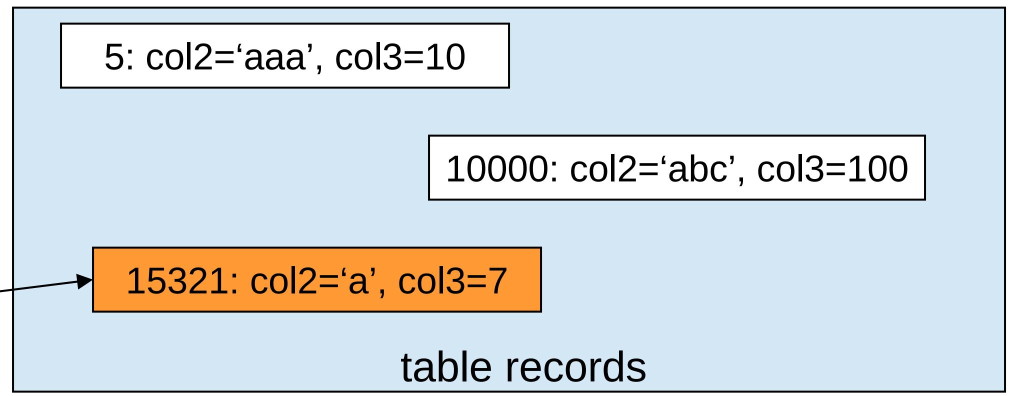
```
> explain select count(c) from t
           id: 1
  select_type: SIMPLE
        table: t
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 100000181
        Extra:

mysql> select count(c) from d;
+-----------+
| count(c)  |
+-----------+
| 100000000 |
+-----------+
1 row in set (28.99 sec)
```

Multi column index

SELECT * FROM tbl WHERE keypart1 = 2 AND keypart2 = 3

Leaf Block 1		
keypart1	keypart2	RowID
1	5	10000
2	1	5
2	2	4
2	3	15321
3	1	100
3	2	200
3	3	300
4	1	400

…



- one access for leaf block, one access for the table records

(If keypart2 is not indexed, three random accesses happen for the table records)

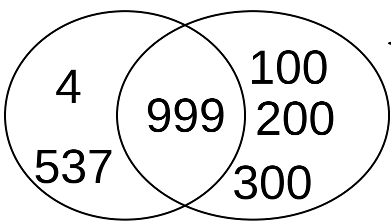
Index merge

SELECT * FROM tbl WHERE key1 = 2 AND key2 = 3

Key1's Leaf Block	
key1	RowID
1	10000
2	4
2	537
2	999
3	100
3	200
3	300
4	400

…

merge



Key2's Leaf Block	
key2	RowID
1	10
1	20
1	30
2	500
3	100
3	200
3	300
3	999

5: col2='aaa', col3=10

999: col2='a', col3=7

table records

- Key 1 and Key2 are different indexes each other
- One access for key1, One access for key2, merging 7 entries, one access on the data
- The more records matched, the more overhead is added

Case: index merge vs multi-column index

```
mysql> SELECT count(*) as c FROM t WHERE t.type=8 AND t.member_id=10;
+---+
| 2 |
+---+
1 row in set (0.06 sec)
         type: index_merge
          key: idx_type,idx_member
      key_len: 2,5
         rows: 83
 Extra: Using intersect(idx_type,idx_member); Using where; Using index
```

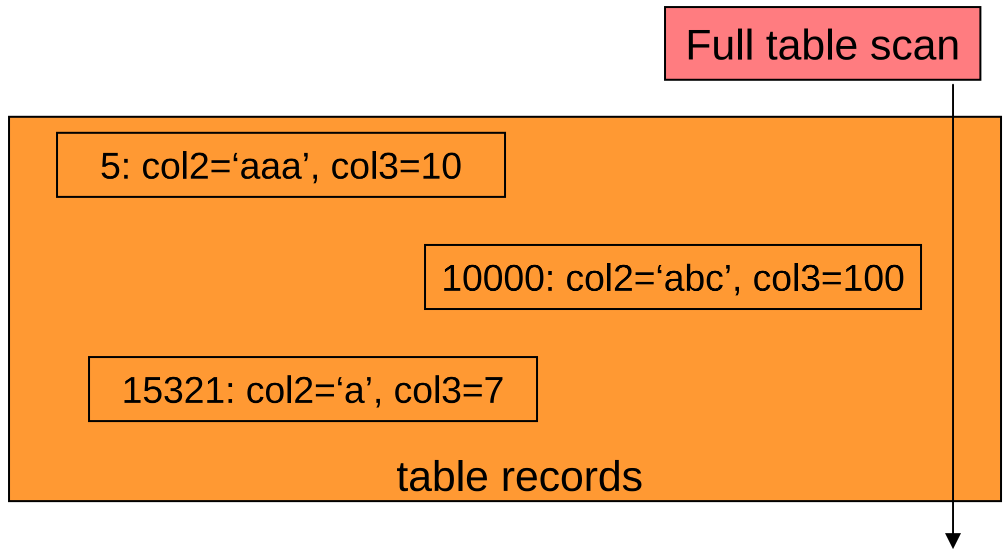
```
mysql> ALTER TABLE t ADD INDEX idx_type_member(type,member_id);
mysql> SELECT count(*) as c FROM t WHERE t.type=8 AND t.member_id=10;
+---+
| 2 |
+---+
1 row in set (0.00 sec)
         type: ref
          key: idx_type_member
      key_len: 7
          ref: const,const
         rows: 1
        Extra: Using index
```

Cases when multi-column index can not be used

```
SELECT * FROM tbl WHERE keypart2 = 3

SELECT * FROM tbl WHERE keypart1 = 1 OR keypart2 = 3
```

Leaf 1		
keypart1	keypart2	RowID
1	5	10000
2	1	5
2	2	4
2	3	15321
3	1	100
3	2	200
3	3	300
4	1	400

…



- The first index column must be used
- Index Skip Scan is not supported in MySQL

Range scan & multi-column index

SELECT * FROM tbl WHERE keypart1 < '2009-03-31 20:00:00' AND keypart2 = 3 ;

Leaf 1		
keypart1	keypart2	RowID
2009-03-29 01:11:11	5	10000
2009-03-29 02:11:11	10	5
2009-03-29 03:11:11	1	4
2009-03-30 04:11:11	3	999
2009-03-31 01:11:11	7	100
2009-03-31 02:11:11	4	200
2009-03-31 03:11:11	1	300
2009-04-01 01:11:11	2	400

Leaf 1		
keypart2	keypart1	RowID
1	2009-03-29 03:11:11	4
1	2009-03-31 03:11:11	300
2	2009-04-01 01:11:11	400
3	2009-03-30 04:11:11	999
3	2009-04-29 02:11:11	6
4	2009-03-31 02:11:11	200
5	2009-03-29 01:11:11	10000
7	2009-03-31 01:11:11	100

5: col2='aaa', col3=10

10000: col2='abc', col3=100

999: col2='a', col3=7

table records

- keypart2 is sorted by keypart1
- If cardinality of keypart1 is very high (common for DATETIME/TIMESTAMP) or used with range scan, keypart2 is useless to narrow records

Covering index & multi-column index

SELECT a,b FROM tbl WHERE secondary_key < 100;

Leaf 1	
sec_key	RowID
1	4
1	10000
1	5
1	15321
1	100
2	200
3	300
..	…

5: status=1, col2='aaa', col3=10

10000: status=0, col2='abc', col3=100

15321: keypart2=3, col2='a', col3=7

table records

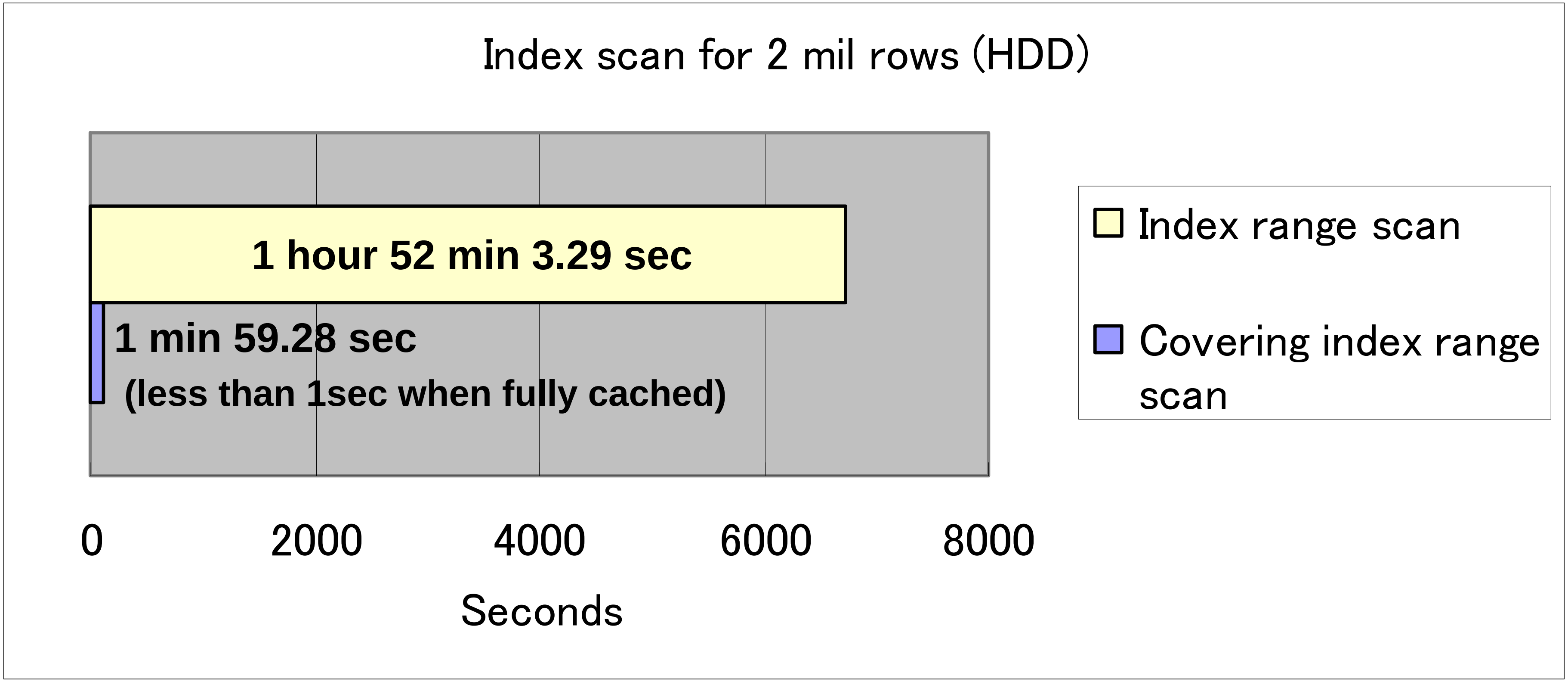
Leaf 1			
sec_key	a	b	RowID
1	2009-03-29	1	4
1	2009-03-30	0	10000
1	2009-03-31	1	5
1	2009-04-01	1	15321
1	2009-03-31	1	100
2	2009-03-30	2	200
3	2009-04-13	3	300
..	…	..	400

- {a, b} are not useful to narrow records, but useful to do covering index
- In InnoDB, PK is included in secondary indexes

Back to the previous benchmark..

- Query: SELECT * FROM tbl WHERE seconday_key < 2000000
- Can be done by covering index scan
- ALTER TABLE tbl ADD INDEX (secondary_key, a, b, c..) ;

Benchmarks: index range vs covering index range



- Warning: Adding additional index has side effects
  - Lower write performance
  - Bigger index size
  - Lower cache efficiency

Speeding up Inserts

What happens when inserting

INSERT INTO tbl (key1) VALUES (61)

Leaf Block 1	
key1	RowID
1	10000
2	5
3	15321
…	
60	431

Leaf block is (almost) full

Leaf Block 1	
key1	RowID
1	10000
2	5
3	15321
…	
60	431

Leaf Block 2	
key1	RowID
61	15322
Empty	

A new block is allocated

Sequential order INSERT

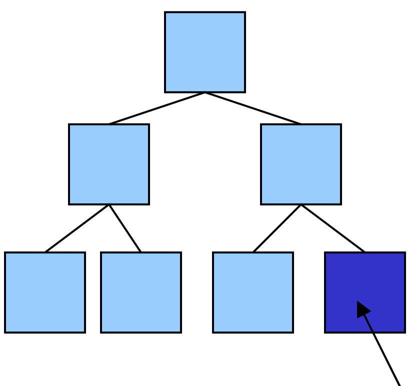
INSERT INTO tbl (key1) VALUES (current_date())

Leaf Block 1	
key1	RowID
2008-08-01	1
2008-08-02	2
2008-08-03	3
…	
2008-10-29	60

Leaf Block 1	
key1	RowID
2008-08-01	1
2008-08-02	2
2008-08-03	3
…	
2008-10-29	60

Leaf Block 2	
key1	RowID
2008-10-29	61
Empty	

- Some indexes are inserted by sequential order (i.e. auto_increment, current_datetime)
- Sequentially stored
- No fragmentation
- Small number of blocks, small size
- Highly recommended for InnoDB PRIMARY KEY



All entries are inserted here: cached in memory

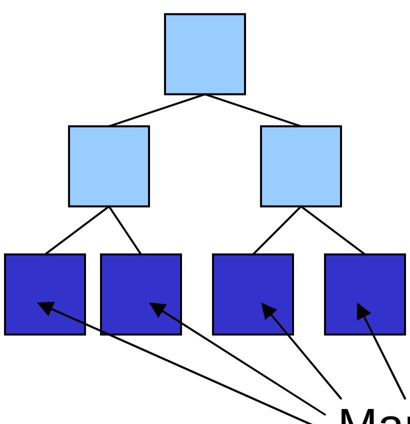
Random order INSERT

INSERT INTO message_table (user_id) VALUES (31)

Leaf Block 1	
user_id	RowID
1	10000
2	5
3	15321
…	
60	431

Leaf Block 1	
user_id	RowID
1	10000
…	
30	333
Empty	

Leaf Block 2	
user_id	RowID
31	345
…	
60	431
Empty	



- Normally insert ordering is random (i.e. user_id on message_table)

- Fragmentated
- Small number of entries per each leaf block
- More blocks, bigger size, less cached

Many leaf blocks are modified: less cached in memory

Random order INSERT does read() for indexes

INSERT INTO message (user_id) VALUES (31)

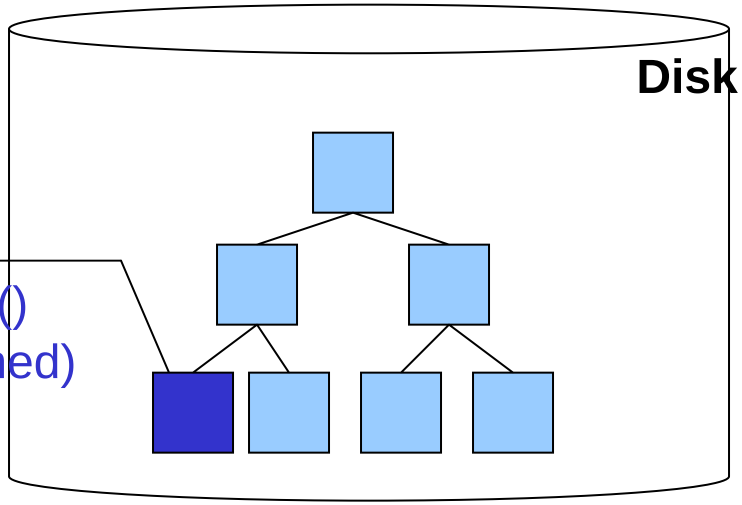
1. Check indexes are cached or not

3. Modify indexes

Buffer Pool

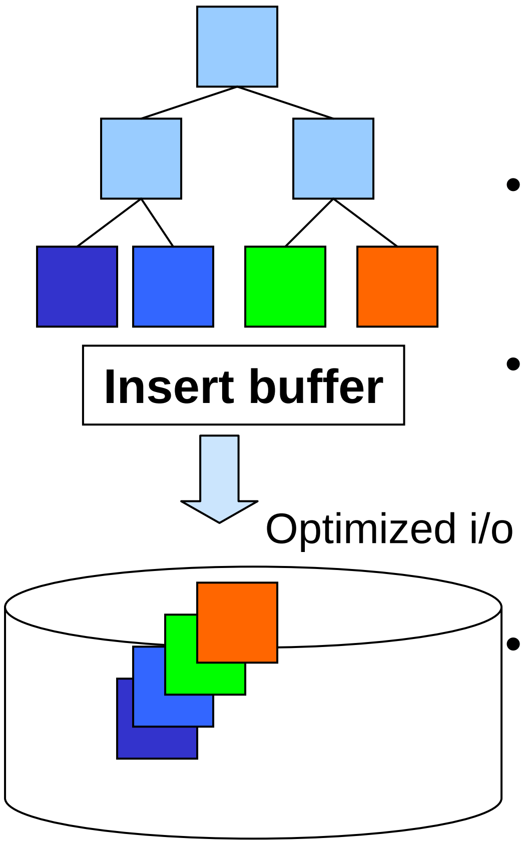
Leaf Block 1	
user_id	RowID
1	10000
2	5
3	15321
…	
60	431

2. pread()
(if not cached)



- Index blocks must be in memory before modifying/inserting index entries
- When cached within RDBMS buffer pool, pread() is not called. Otherwise pread() is called
- Sequentially stored indexes (AUTO_INC, datetime, etc) usually do not suffer from this
- Increasing RAM size / using SSD helps to improve write performance

InnoDB feature: Insert Buffering

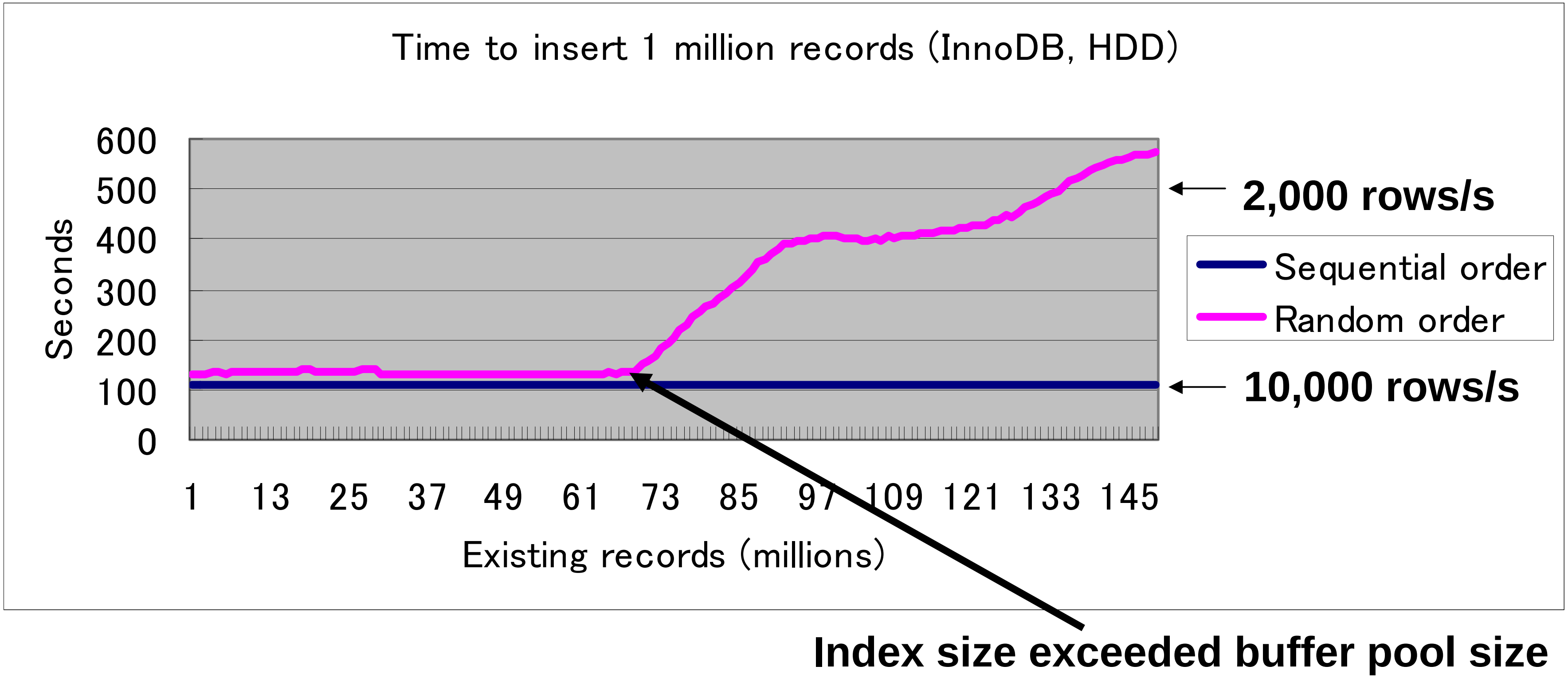


- If non-unique, secondary index blocks are not in memory, InnoDB inserts entries to a special buffer(“insert buffer”) to avoid random disk i/o operations
  - Insert buffer is allocated on both memory and innodb SYSTEM tablespace
- Periodically, the insert buffer is merged into the secondary index trees in the database (“merge”)
- Pros: Reducing I/O overhead
  - Reducing the number of disk i/o operations by merging i/o requests to the same block
  - Some random i/o operations can be sequential
- Cons:
  Additional operations are added
  Merging might take a very long time
  - when many secondary indexes must be updated and many rows have been inserted.
  - it may continue to happen after a server shutdown and restart

Benchmarks: Insert performance

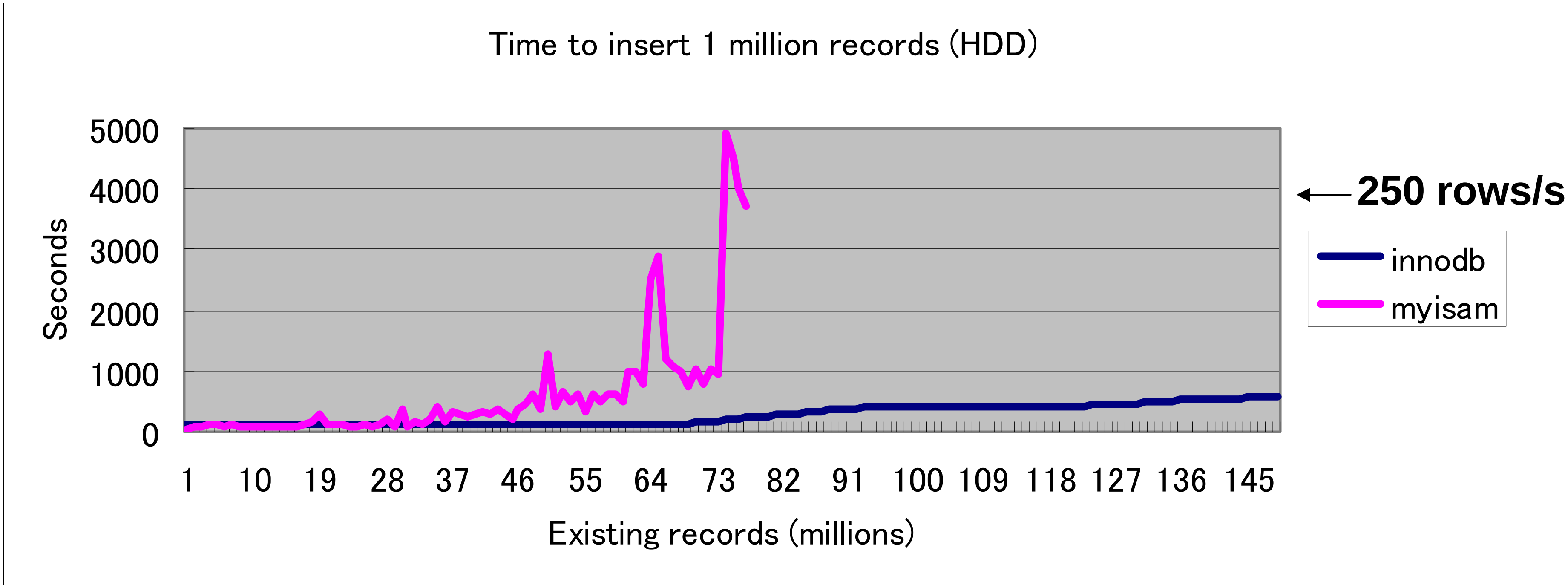
- Inserting hundreds of millions of records
  - Suppose high-volume of insert table (twitter mesasge, etc..)
- Checking time to add one million records
- Having three secondary indexes
  - Random order inserts vs Sequential order inserts
    - Random: INSERT .. VALUES (id, rand(), rand(), rand());
    - Sequential: INSERT .. VALUES (id, id, id, id)
  - Primary key index is AUTO_INCREMENT
- InnoDB vs MyISAM
  - InnoDB: buffer pool=5G, O_DIRECT, trx_commit=1
  - MyISAM: key buffer=2G, filesystem cache=5G
- Three indexes vs One index
- Changing buffer pool size
- 5.1 Partitioning or not

Benchmarks (1) : Sequential order vs random order



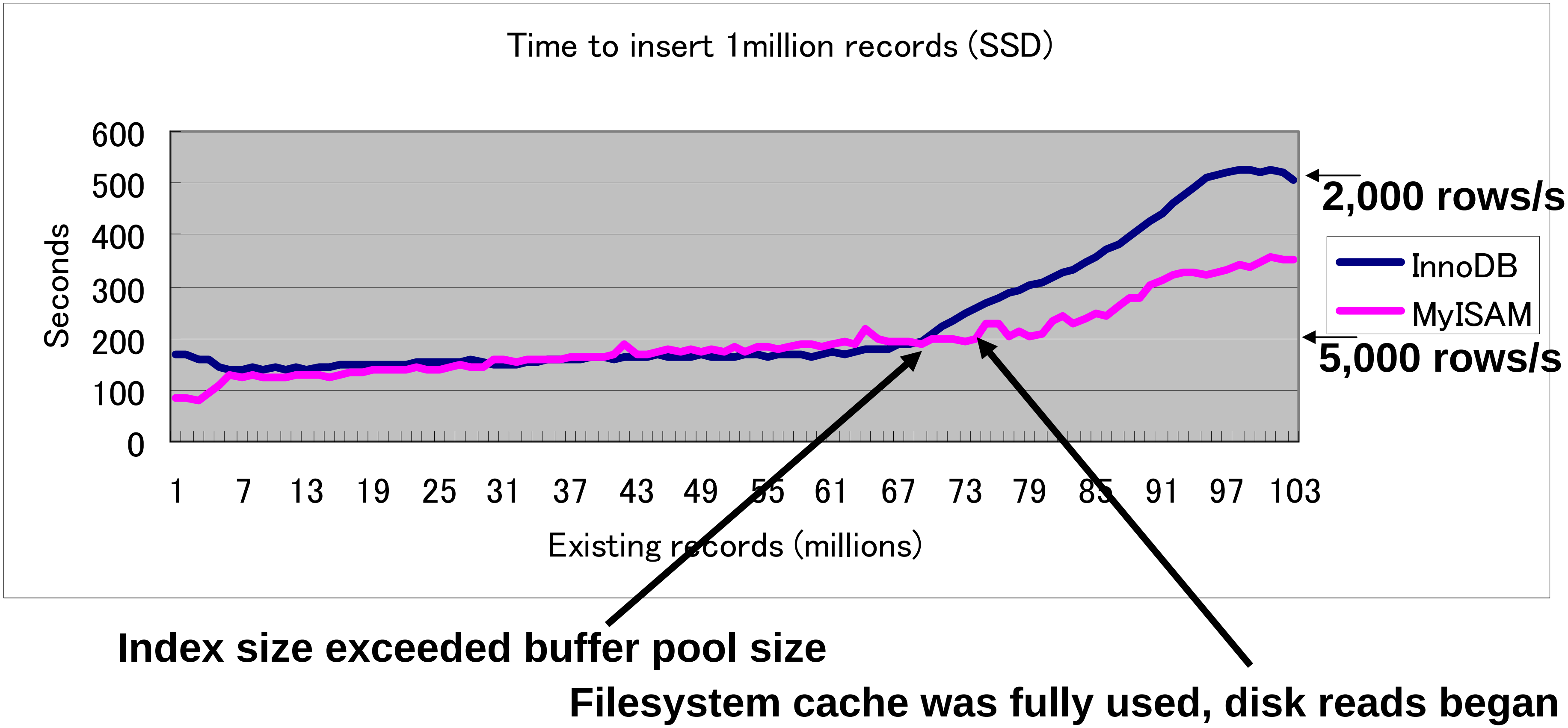
- Index size exceeded innodb buffer pool size at 73 million records for random order test
- Gradually taking more time because buffer pool hit ratio is getting worse (more random disk reads are needed)
- For sequential order inserts, insertion time did not change.
  No random reads/writes

Benchmarks (2) : InnoDB vs MyISAM (HDD)



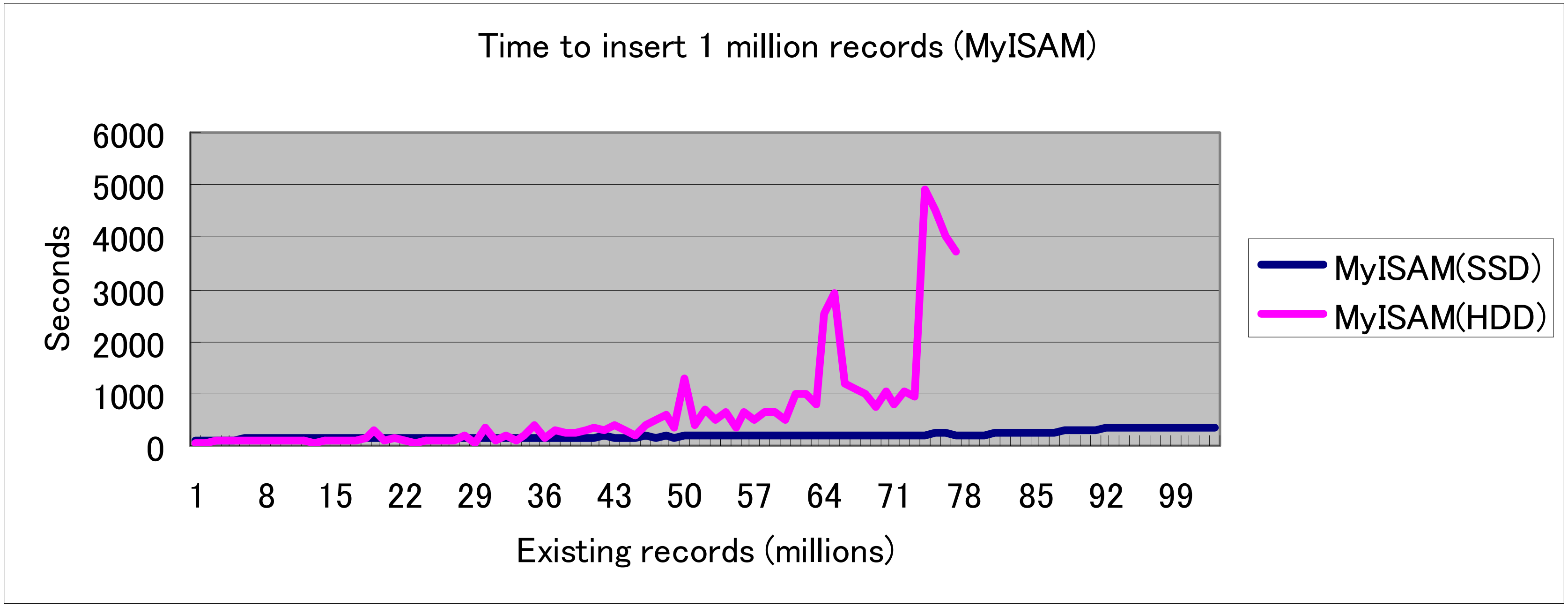
- MyISAM doesn't do any special i/o optimization like "Insert Buffering" so a lot of random reads/writes happen, and highly depending on OS
- Disk seek & rotation overhead is really serious on HDD

Benchmarks(3) : MyISAM vs InnoDB (SSD)



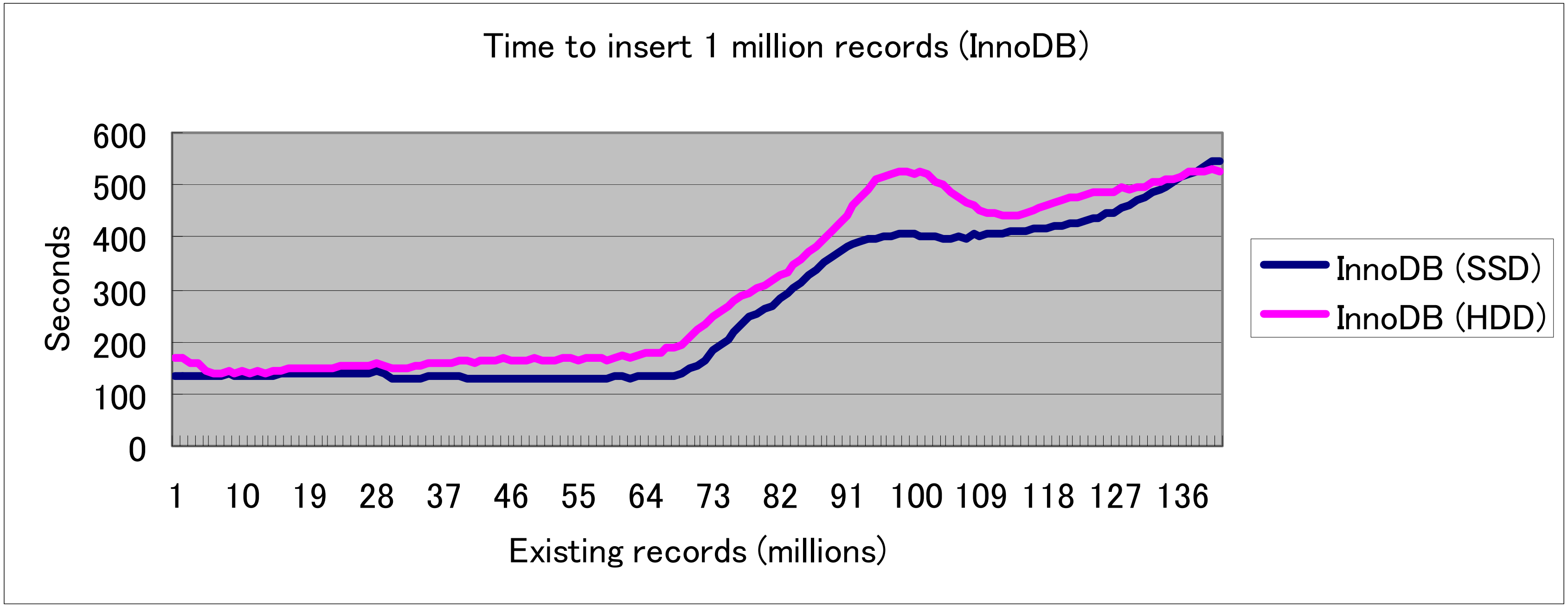
- MyISAM got much faster by just replacing HDD with SSD !

Benchmarks (4) : SSD vs HDD (MyISAM)



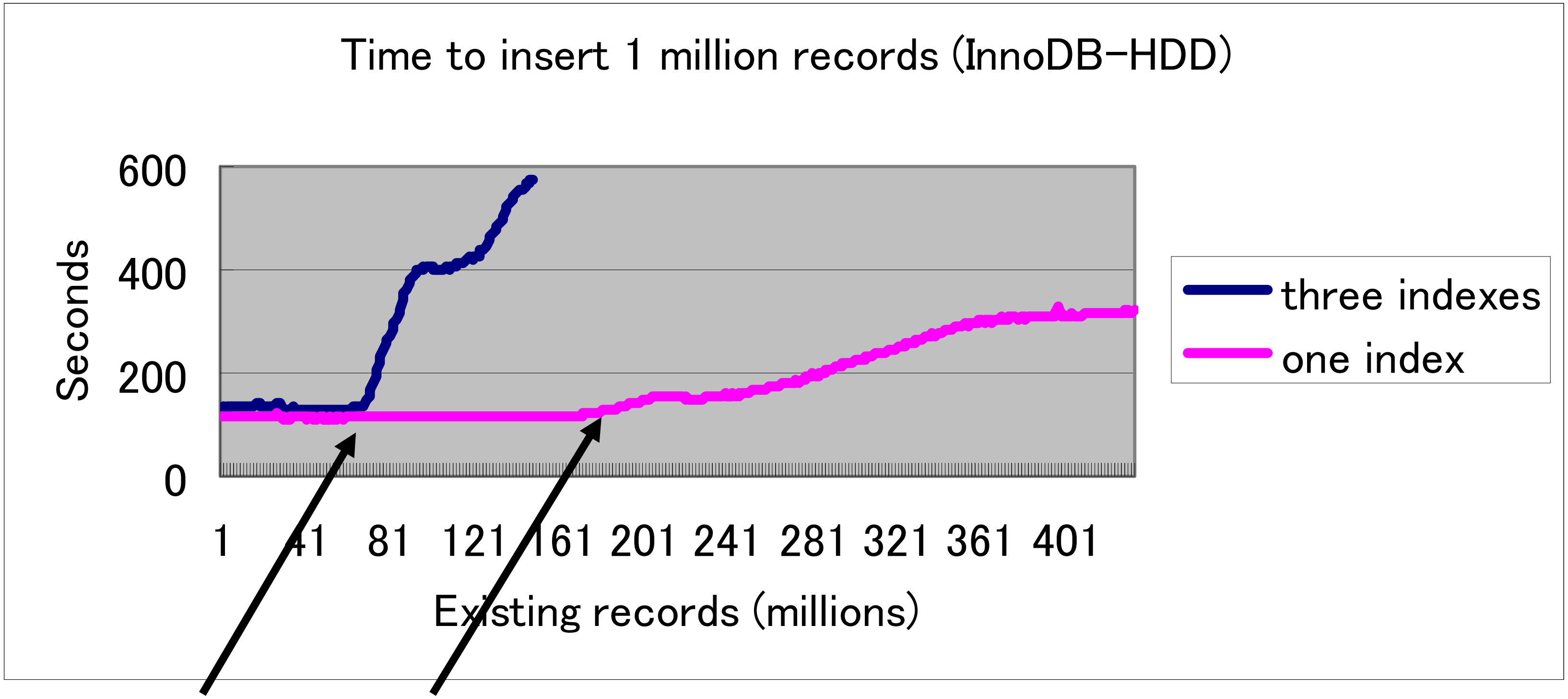
- MyISAM on SSD is much faster than on HDD
- No seek/rotation happens on SSD

Benchmarks (5) : SSD vs HDD (InnoDB)



- SSD is 10% or more faster
- Not so big difference because InnoDB insert buffering is highly optimized for HDD
- Time to complete insert buffer merging was three times faster on SSD (SSD:15min / HDD:42min)

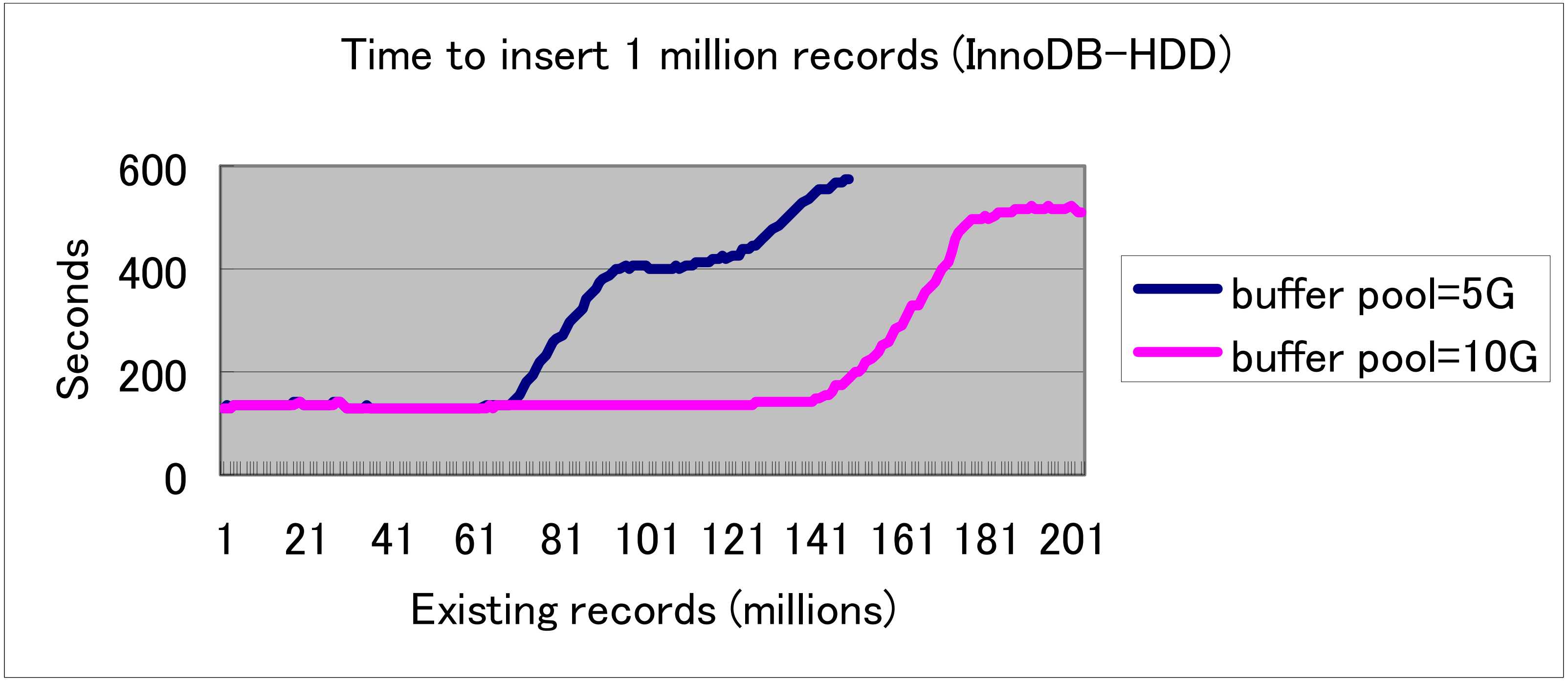
Benchmarks (6) : Three indexes vs Single index



Index size exceeded buffer pool size at these points

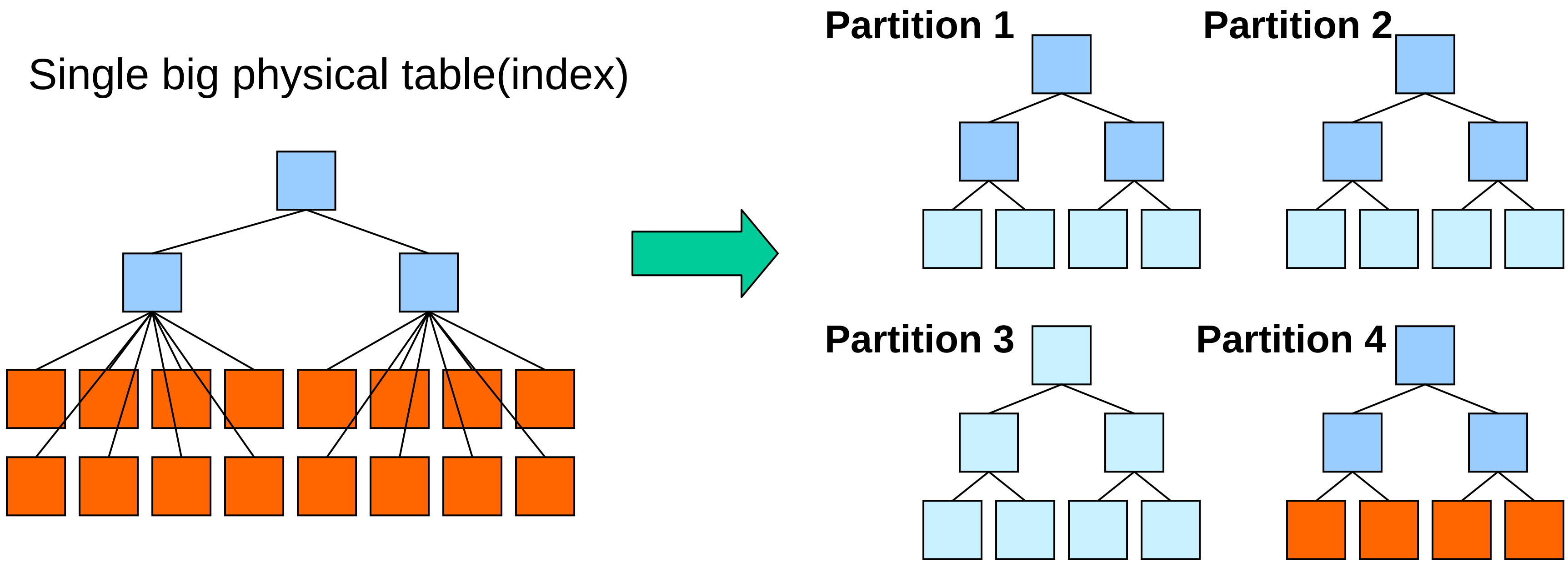
- For single index, index size was three times smaller so exceeded buffer pool size much slowly
- For single index, random i/o overhead was much smaller
- Common Practice: Do not create unneeded indexes

Benchmarks (7) : Increasing RAM (InnoDB)



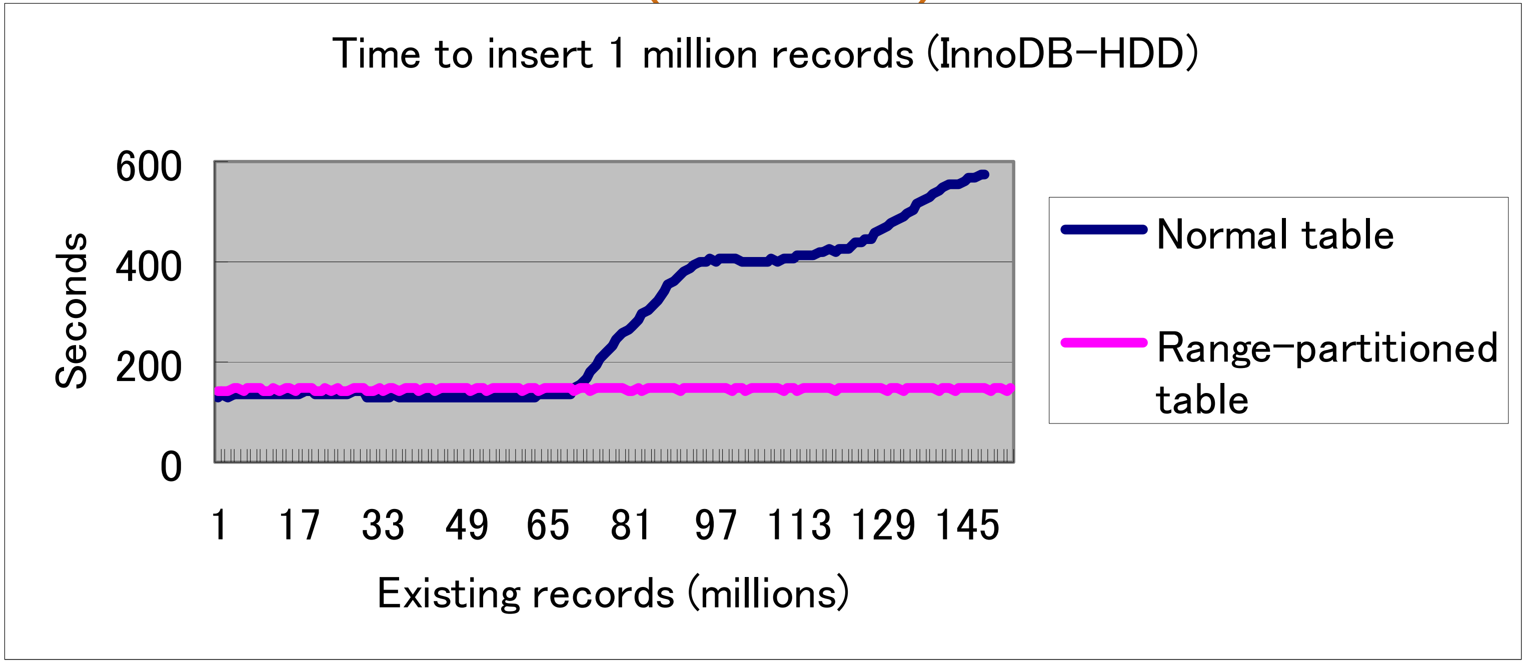
- Increasing RAM (allocating more memory for buffer pool) raises break-even point
- Common practice: Make index size smaller

Make index size smaller



- When all indexes (active/hot index blocks) fit in memory, inserts are fast
- Follow the best practices to decrease data size (optimal data types, etc)
- Sharding
- MySQL 5.1 range partitioning
  - Range Partitioning , partitioned by sequentially inserted column (i.e. auto_inc id, current_datetime)
  - Indexes are automatically partitioned
  - Only index blocks in the latest partition are *hot* if old entries are not accessed

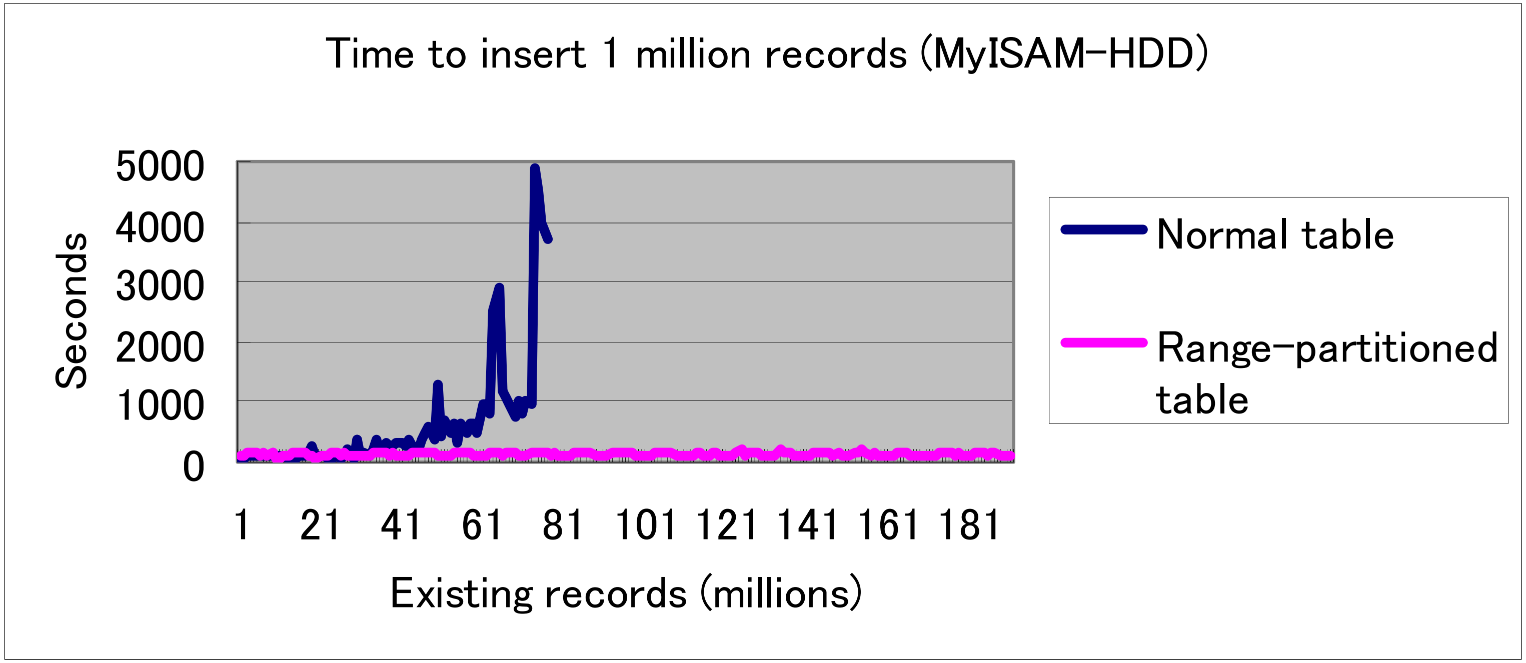
Benchmarks(8) : Using 5.1 Range Partitioning (InnoDB)




```
PARTITION BY RANGE(id) (
    PARTITION p1 VALUES LESS THAN (10000000),
    PARTITION p2 VALUES LESS THAN (20000000),
    ....
```

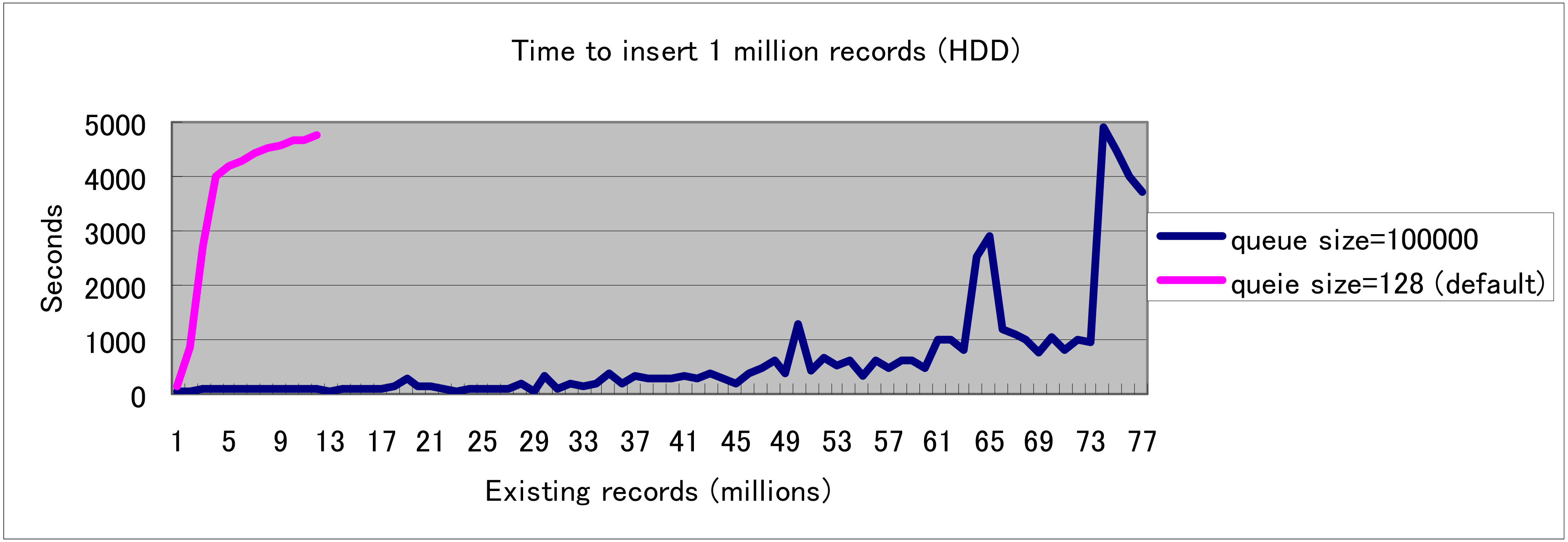
- On insert-only tables (logging/auditing/timeline..), only the latest partition is updated.

Benchmarks(9) : Using 5.1 Range Partitioning (MyISAM)



- Random read/write overhead is small for small indexes
  - No random read when fitting in memory
  - Less seek overhead for small indexes

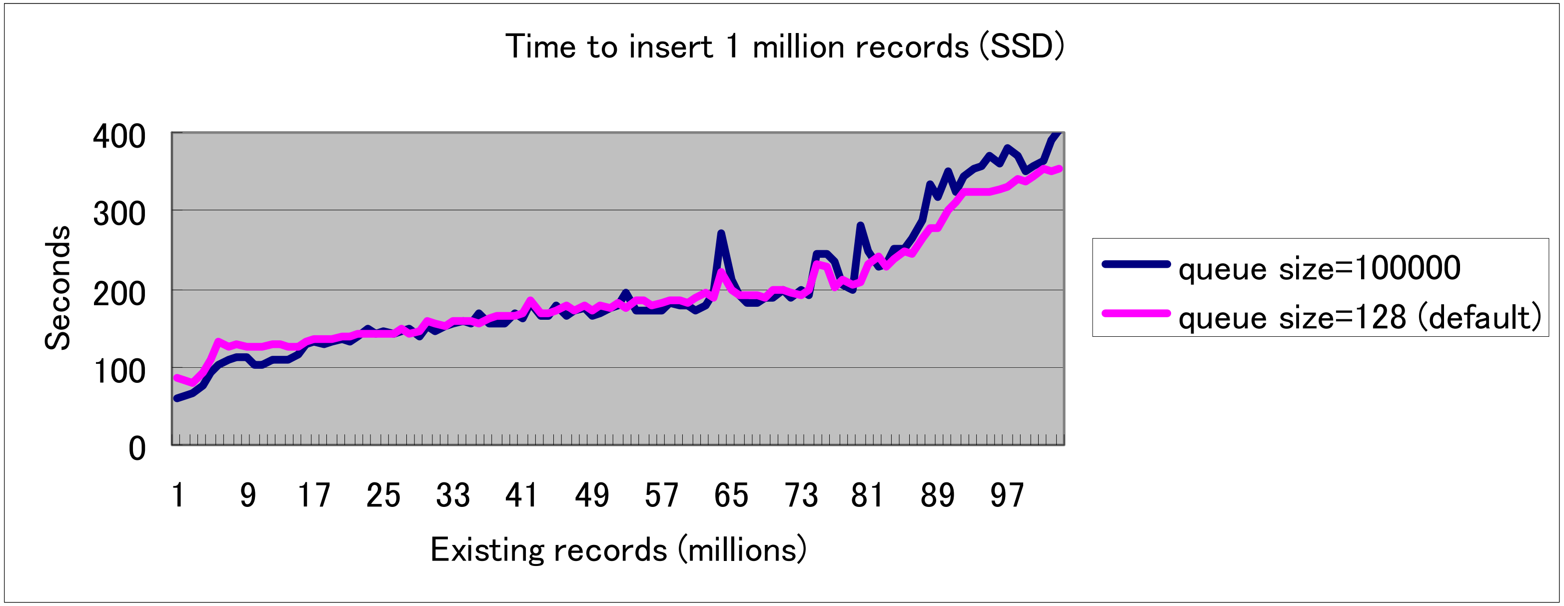
Benchmarks (10) : Linux I/O Scheduler (MyISAM-HDD)



- MyISAM doesn’t have mechanism like insert buffer in InnoDB
  - I/O optimization highly depends on OS and storage
- Linux I/O scheduler has an “i/o request queue”
- Sorting requests in the queue to process i/o effectively
- Queue size is configurable

```
# echo 100000 > /sys/block/sdX/queue/nr_requests
```

Benchmarks (11) : Linux I/O Scheduler (MyISAM-SSD)



- No big difference
- Sorting i/o requests decreases seek & rotation overhead for HDD, but not needed for SSD

Summary

- Minimizing the number of random access to disks is very important to boost index performance
  - Increasing RAM boosts both SELECT & INSERT performance
  - SSD can handle 10 times or more random reads compared to HDD
- Utilize common & MySQL specific indexing techniques
  - Do not create unneeded indexes (write performance goes down)
  - covering index for range scans
  - Multi-column index with covering index
- Check Query Execution Plans
  - Control execution plans by FORCE/IGNORE INDEX if needed
  - Query Analyzer is very helpful for analyzing
- Create small-sized indexes
  - Fitting in memory is very important for random inserts
  - MySQL 5.1 range partitioning helps in some cases
- MyISAM is slow for inserting into large tables (on HDD)
  - when using indexes and inserting by random order
  - "MyISAM for Logging/Auditing table" is not always good
  - Using SSD boosts performance

Enjoy the conference !