インデックスを使いこなす

Yoshinori Matsunobu

Senior MySQL Consultant
Professional Services APAC
Sun Microsystems
Yoshinori.Matsunobu@sun.com

自己紹介

- 松信 嘉範 (まつのぶ よしのり)

- 2006年9月からMySQLシニアコンサルタントとして勤務
  - パフォーマンスチューニング、環境レビュー、MySQL Cluster、ベストプラクティス等
  - APAC (日本およびアジア圏)を主担当
    - 日本、オーストラリア、ニュージーランド、シンガポール、インド、香港など
  - コンサルティング依頼を常時受付中

- 対外的な活動
  - 書籍(自著)
    - 現場で使えるMySQL (2006.3 翔泳社)
    - Javaデータアクセス実践講座 (2008.2 翔泳社)
  - 連載
    - Linux-DBサーバ構築入門 (翔泳社 DBマガジン)
  - 執筆依頼を随時受付中
  - 日程の事前確保が難しいので、勉強会/セミナー系よりも執筆系がメイン

Copyright 2008 Sun Microsystems inc

本セッションの内容

- 検索処理とインデックス
  - B+Treeインデックスの構造
  - Covering Index
  - インデックス検索とフルテーブルスキャン
  - マルチカラムインデックスとインデックスマージ
  - ソート処理とインデックス
  - ケーススタディ: DBT-1
- 更新処理とインデックス
  - INSERTをすると何が起こるのか
  - 昇順INSERTとランダムINSERT
  - 昇順INSERTのためのアプローチ
- プラクティス (時間があれば)
- MyISAMインデックスとLinux I/Oスケジューラ (時間があれば)

ディスクI/O性能を意識する

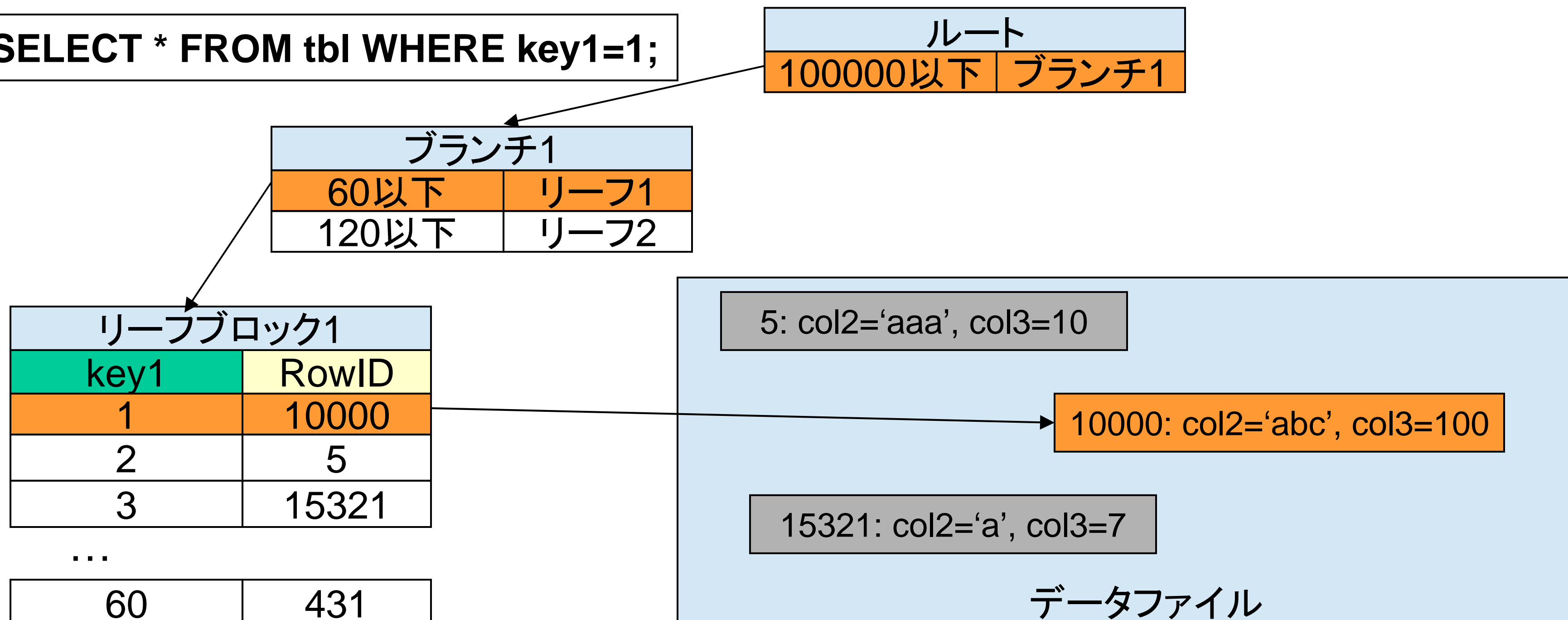
CPUのアクセス時間: < 10ns
メモリのアクセス時間: < 60ns
HDDのアクセス時間: < 5ms
* シーク待ち+回転待ち+転送時間
SSDのアクセス時間: < 0.5ms

HDDのアクセス時間が支配的
HDD+ライトキャッシュ+バッテリーバックアップで
書き込みは大きく緩和できるが、
読み取りは緩和できない

HDD→SSDで読み取りが大きく改善される
今後の普及に大きな期待

Copyright 2008 Sun Microsystems inc

B+Treeインデックスの構造とI/O



- インデックスの内部では、インデックス列に対して昇順に格納されている
- インデックス検索のたどり方の基本形は、ルート→ブランチ→リーフ→データファイル
- キーに対応する行番号(RowID)から、データファイル上の場所を一意に特定できる
- I/Oの単位はブロック (InnoDBは16KB)
- 1つのブロックの中に数十以上のエントリがあるのが普通
- 読み取られたブロックは、メモリ上にキャッシュされる
- ルート、ブランチはほぼ確実にキャッシュされる
- テーブルが巨大な場合、リーフとデータファイルの中にはキャッシュされないものが出てくる
- 見積もりとしては、リーフとデータファイルで計2回のランダムディスクアクセスが起きると考える

Copyright 2008 Sun Microsystems inc

InnoDBのインデックス構造

セカンダリインデックス
(主キー以外)

リーフブロック1	
key1	PK
1	10
2	1
3	3
…	
60	5

クラスタインデックス
(主キー)

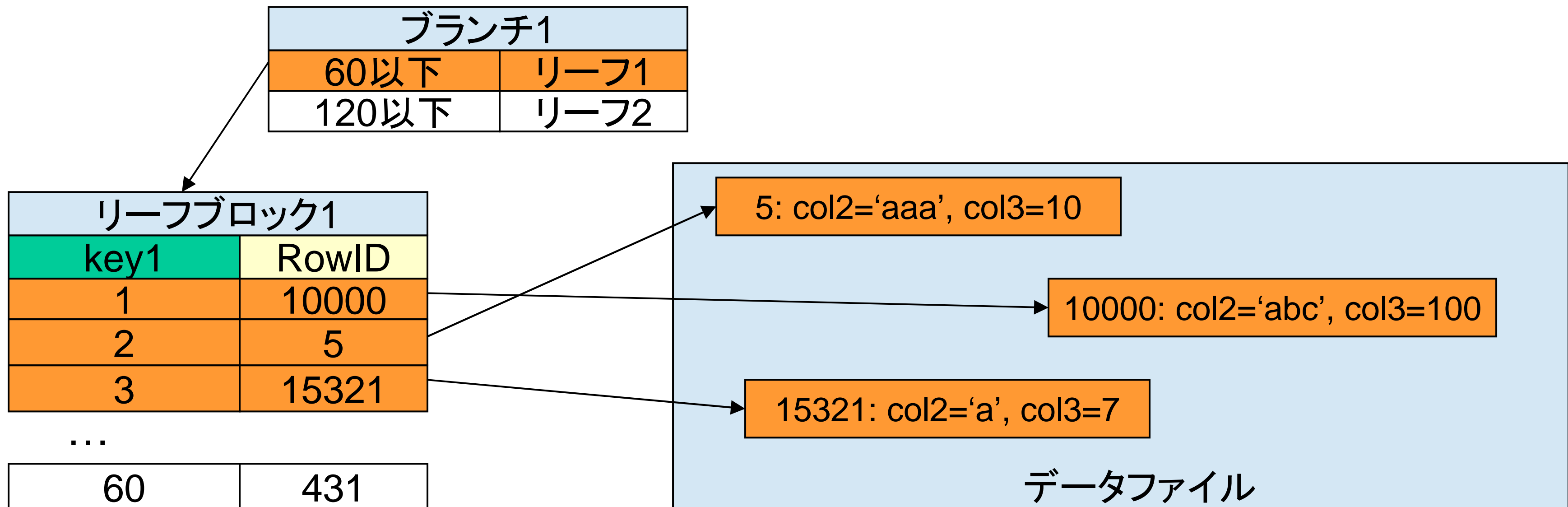
リーフブロック1	
PK	残りの列の値
1	Col2='a', col3='2008-10-31'….
2	Col2='aaa', col3='2008-10-31'….
3	Col2='aaa', col3='2008-10-31'….
…	
10	Col2='abc', col3='2008-10-31'….

・主キー以外のインデックス (セカンダリインデックス) には、インデックス値に対して
主キー値が対応 (RowIDではない)
・主キーインデックス (クラスタインデックス) には、主キー値に対して
残りの列値が対応 (RowIDではない)
・主キー検索では、1回のアクセスで全列値を取ることができるので高速
・主キー以外のインデックスでの検索は、セカンダリインデックスにアクセスした後、
クラスタインデックスにアクセスする必要があるのでオーバーヘッドが大きい
・可能な限り主キー検索を行なうようにする
・可能な限り主キーは小さくする (INTEGER UNSIGNEDなど)
・クラスタインデックスは、セカンダリインデックスに比べてサイズが大きくなる

Copyright 2008 Sun Microsystems inc

B+Treeインデックスの構造とI/O(範囲検索)

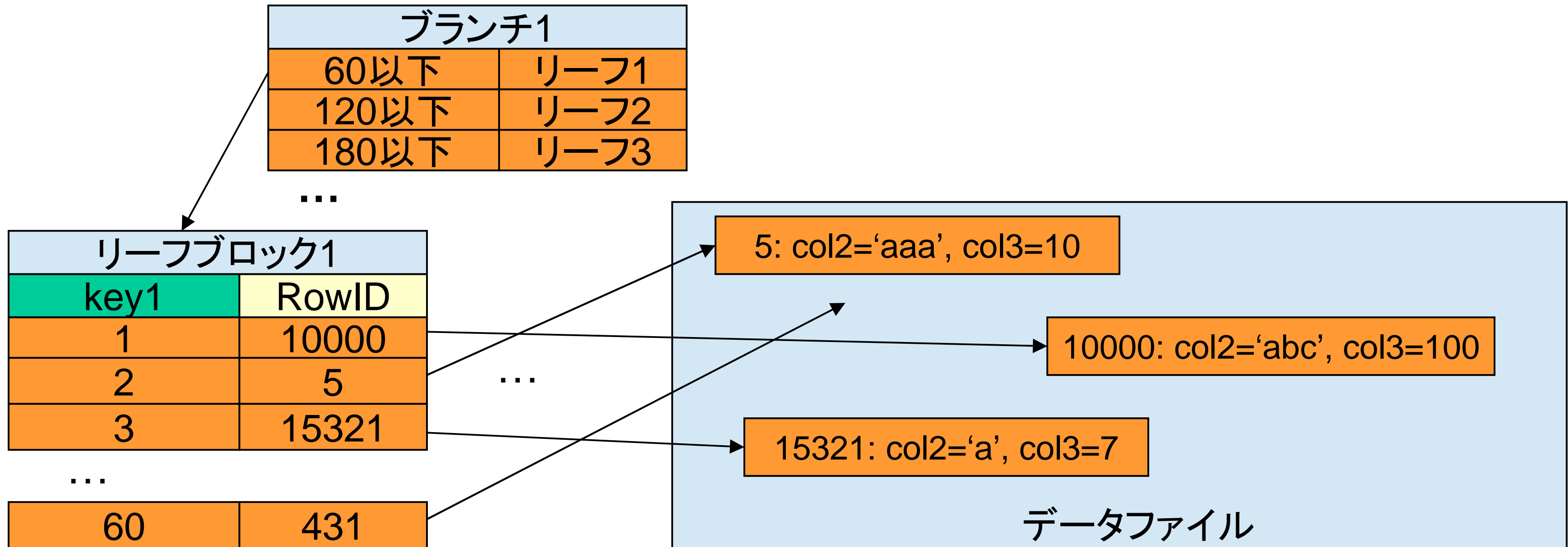
SELECT * FROM tbl WHERE key1 IN (1, 2, 3);



- 通常の用途では、1つのブロックの中に数十以上のエントリがある
- したがって、key1の1,2,3は1つのブロック内におさまっていると考えられる　→1回のI/O
- 対応するRowIDはばらばらなので、データファイル上の配置も飛び飛び
  →各レコードを読むのにそれぞれ1回のI/Oが必要と考えられる
- 見積もりとしては、リーフ1回、データファイル3回の計4回のランダムリードが発生
- 一般化すると、N件の範囲検索では、1 + N回のランダムリードが発生しうる

Copyright 2008 Sun Microsystems inc

インデックスの副作用



・インデックス検索は、データファイルを読みに行く処理がランダムアクセスになる
・インデックスにヒットした件数だけデータファイルをランダムアクセスする
・100万件インデックスにヒットすれば、100万回ランダムアクセスする
・RDBMSの(コストベース)オプティマイザは、通常はこのような検索方式を選択せず、フルテーブルスキャンを選択する

Copyright 2008 Sun Microsystems inc

フルテーブルスキャン

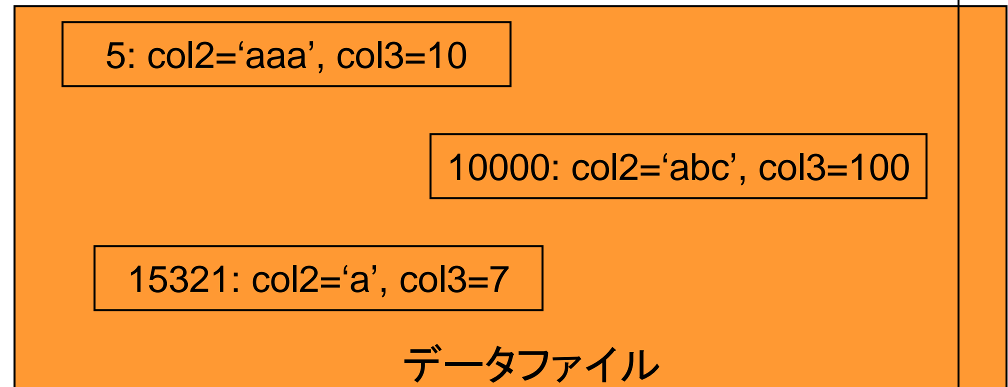
SELECT * FROM tbl WHERE key1 <= 1000000

ブランチ1	
60以下	リーフ1
120以下	リーフ2
120以下	リーフ3

リーフブロック1	
key1	RowID
1	10000
2	5
3	15321
…	
60	431

…

フルテーブルスキャン



- フルテーブルスキャンは、テーブルの先頭から末尾までを順番に読んでいく
- I/O回数は100万回よりもずっと少なくなる
  - 1つのブロックには複数のレコードがあるので、レコード数が100万でもブロック数はもっと少ない
  - RDBMSの「先読み」機能により、フルテーブルスキャンでは複数のブロックを一度に読む
- このため、インデックススキャンよりもずっと効率が良い
- 定説：アクセス範囲が10-15%以上になる場合はフルテーブルスキャンの方が効率が良い

Covering Index (インデックス「だけ」を読む検索)

```
SELECT key1 FROM tbl WHERE key1 IN (1, 2, 3);
```

ブランチ1	
60以下	リーフ1
120以下	リーフ2

リーフブロック1	
key1	RowID
1	10000
2	5
3	15321
…	
60	431

データファイル

- 5: col2='aaa', col3=10
- 10000: col2='abc', col3=100
- 15321: col2='a', col3=7

・Covering Indexとは、データファイルを読まず、インデックスを読むだけで処理を完結できる検索形態のこと
・この場合、リーフブロックを読むだけで処理が完結するので非常に高速
・特に広範囲にまたがるアクセスで威力を発揮
・発生条件は、WHERE句、SELECT句などで指定するすべての列がインデックスに含まれていること
→マルチカラムインデックスで狙いやすい
「SELECT * FROM」がアンチパターンだと言われる理由の1つ

Copyright 2008 Sun Microsystems inc

Covering Indexの見分け方

```
> explain select count(*) from t1
           id: 1
  select_type: SIMPLE
        table: t1
         type: index
possible_keys: NULL
          key: key1
      key_len: 5
          ref: NULL
         rows: 57705
        Extra: Using index
```

```
> explain select count(c3) from t1
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 57705
        Extra:
```

・EXPLAINのExtraに「Using index」があれば、Covering indexになっている
・typeが「range」や「index」の場合、「Using index」を積極的に狙いたい

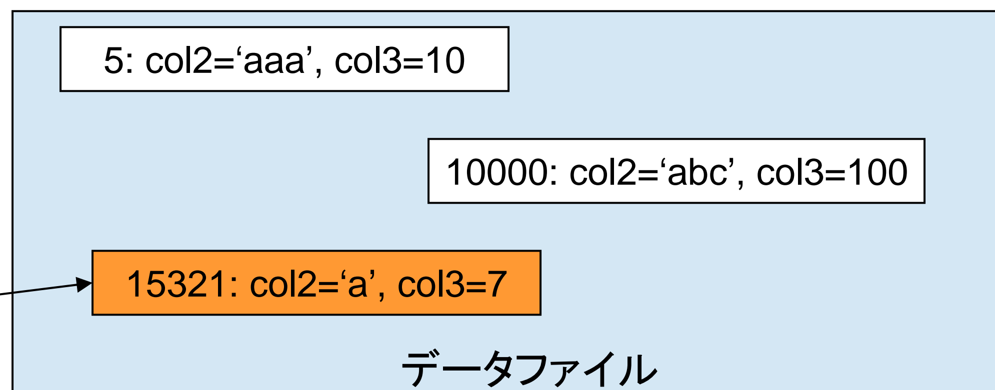
マルチカラムインデックスと
インデックスマージ

マルチカラムインデックス

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

リーフブロック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

…



- keypart1, keypart2の両方がAND条件で指定
- 読み取るリーフブロック数は1個で済む
- マッチしたレコード数が1個なら、データファイルへのアクセスも1回で済む
- Disk Read回数は2回、となり、アクセス効率は悪くない

Copyright 2008 Sun Microsystems inc

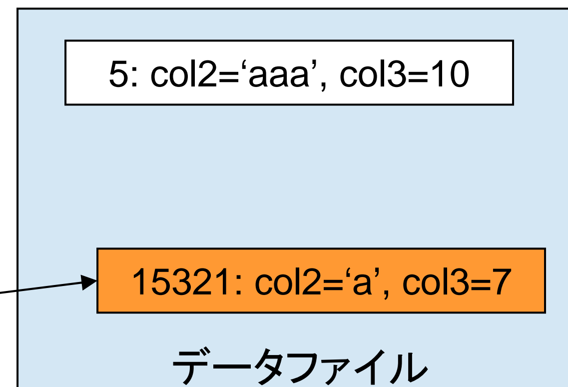
インデックスマージ

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

key1のリーフ	
key1	RowID
1	10000
2	4
2	5
2	15321
3	100
3	200
3	300
4	400

マージ処理

key2のリーフ	
key2	RowID
1	10
1	20
1	30
2	500
2	1000
3	200
3	300
3	15321



…

・key1、key2はそれぞれ別のインデックス
・それぞれのインデックスで条件一致判定
・マッチしたRowIDをそれぞれ比較(マージ)し、RowIDが一致したものが該当
・Disk Readの回数は3回、スキャンしたリーフエントリ数は6、さらにマージ処理が加わり、マルチカラムインデックスよりもオーバーヘッドが大きい
・各インデックスについて、マッチしたレコード数が多ければ多いほど処理に時間がかかる
例：インデックスマージで0.1～0.2秒のクエリが、マルチカラムインデックス化によって0.01秒に

マルチカラムインデックスのきかない検索

```
SELECT * FROM tbl WHERE keypart2 = 3

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

リーフブロック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

…

5: col2='aaa', col3=10

10000: col2='abc', col3=100

15321: col2='a', col3=7

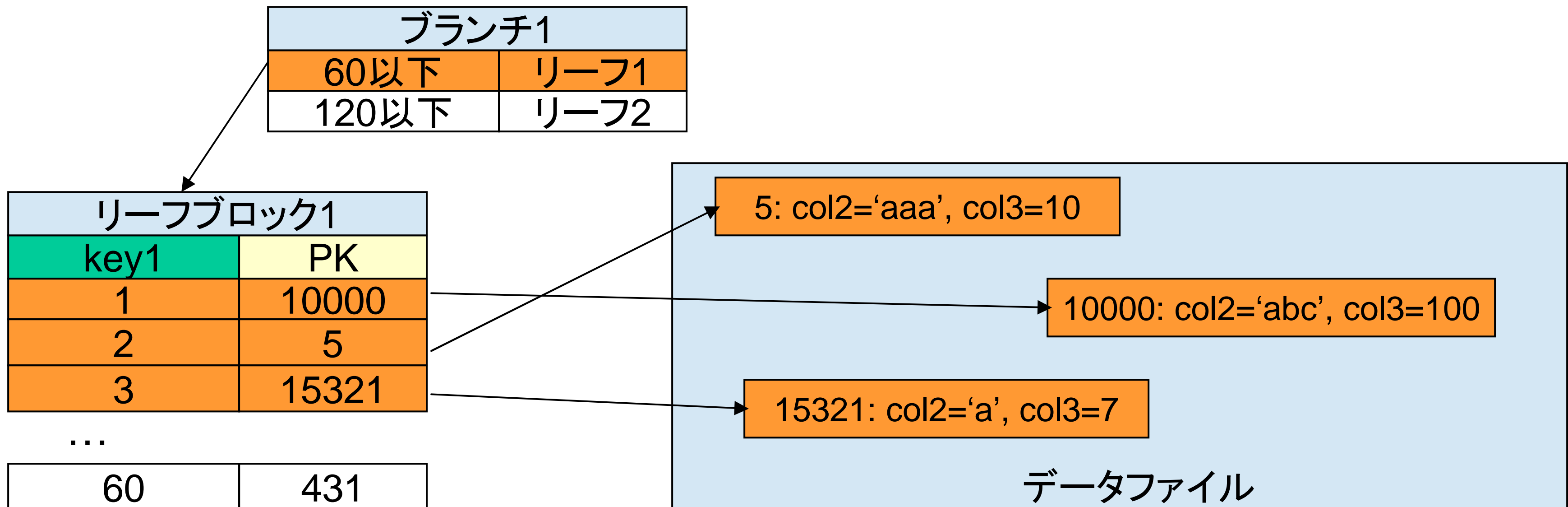
データファイル

・マルチカラムインデックスは、先頭列がWHERE条件で指定されない限り使われない
・OR条件にはきかない
・インデックスマージは、どちらの場合にも効果がある

ソート処理とインデックス

ソート処理とインデックス

SELECT * FROM tbl WHERE key1 < 30 ORDER BY key1

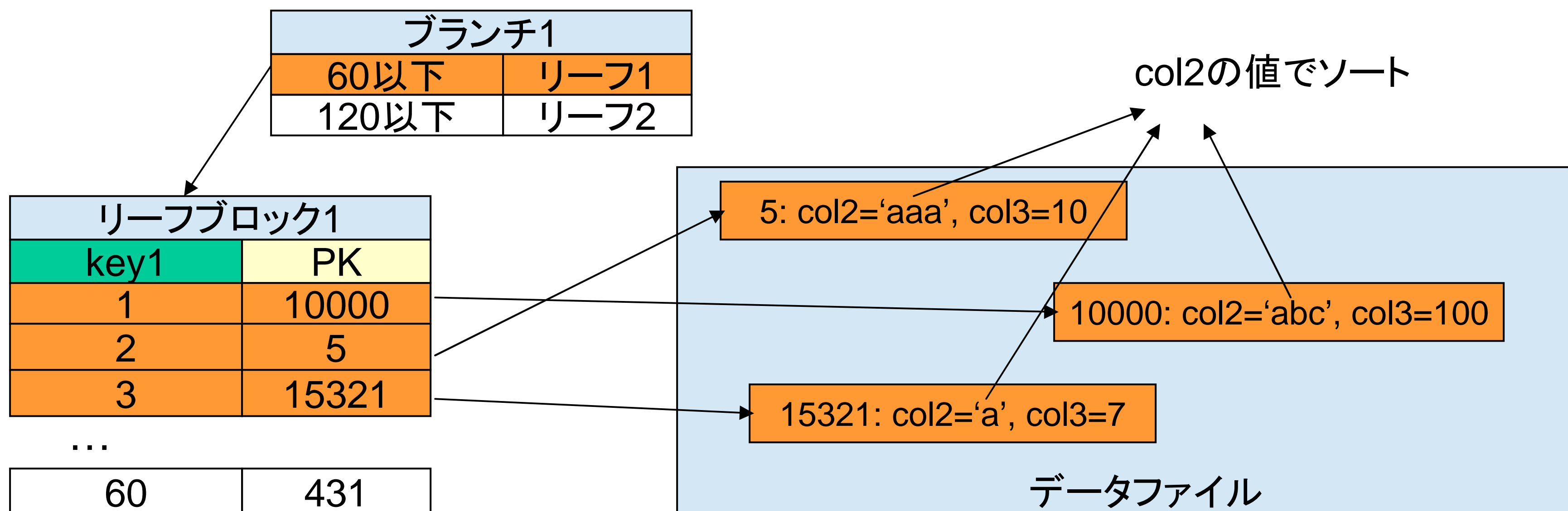


・インデックスはすでにソートされているため、インデックス対象列がソートに使われると、ソート処理そのものが必要ないので高速になる (ソートのオーバーヘッドが無い)

Copyright 2008 Sun Microsystems inc

ソート処理とインデックス (2)

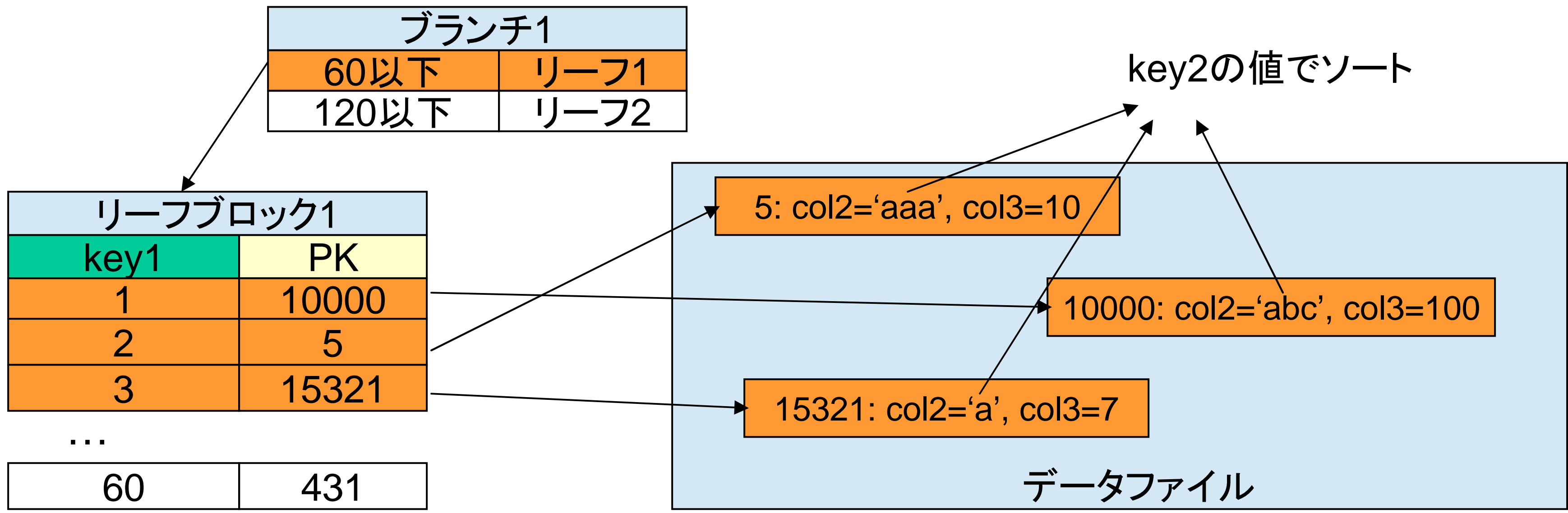
SELECT * FROM tbl WHERE key1 < 30 ORDER BY col2



・ORDER BYの列がインデックス対象でないと、その列でソートをするために結果を並べ替える必要がある。
・EXPLAINのExtraに「Using filesort」と出れば、ソートにインデックスが使われていないことを示している
・ソートしなければいけないレコードが多いほど、時間がかかる

ソート処理とインデックス (3)

SELECT * FROM tbl WHERE key1 < 30 ORDER BY key2



・2つの別々のインデックスがあっても、使われるのはどちらか片方
(この図はkey1が使われる場合を示している)
・key2が使われると「Using filesort」は起きない。しかし、「key1 < 30」の絞込みにインデックスが使えないので、全レコードをスキャンしなければならない
・key1とkey2のどちらのインデックスが使われるかは、場合による
(MySQLのコストベース・オプティマイザによる判断)

ORDER BY LIMIT Nの落とし穴

```
SELECT * FROM tbl WHERE cond < 10000 ORDER BY keyX LIMIT 20
```

- 上位N件を取るために、ORDER BY xxx LIMIT N は頻繁に使われる
- 選択されうる実行計画は、以下の3種類

A: condがインデックスとして使われ、条件を満たしたレコードをソートし、上位20件を取得
(type=range, key=cond, Using filesort)

B: keyXがインデックスとして使われ、cond条件を満たすレコードが20件になったところで終了
(type=index, key=keyX)

C: condにインデックスが無く、フルテーブルスキャンをし、ソートし、上位20件を取得
(type=ALL, key=NULL, Using filesort)

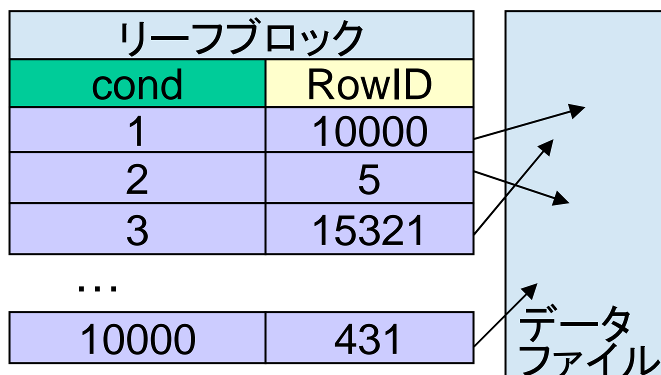
- どれが最も高速になるかは、「場合による」
- 適切な実行計画が選ばれないことがある

Copyright 2008 Sun Microsystems inc

ORDER BY LIMIT Nの落とし穴

SELECT * FROM tbl WHERE cond < 10000 ORDER BY keyX LIMIT 20

A. condをインデックスとして使う

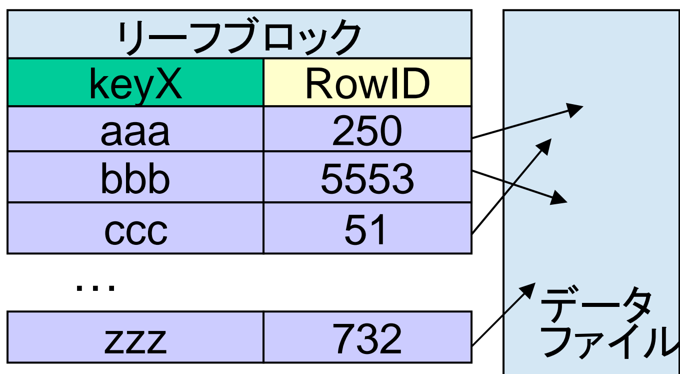


➡ keyXでソート

➡ 上位20件を返す

C. フルテーブルスキャン

B. keyXをインデックスとして使う



➡ Cond< 10000の条件判定
20個満たしたところで終了

Aが最適なケース：
cond < 10000を満たすレコードが
ほとんど無い場合 (大量にあるとだめ)

Bが最適なケース：
cond < 10000を満たすレコードが
大量にある場合 (少ないとだめ)

Cが最適なケース：
condでインデックスが使えず、
Bの条件も満たさない場合

適切な実行計画が選ばれない場合、
FORCE INDEX, IGNORE INDEXヒントでコントロール

ケーススタディ: DBT-1 (Infamous IPA benchmarks)

```
SELECT i_id, i_title, a_fname, a_lname FROM item, author
WHERE i_title LIKE '%BABABAOGBABAIN%' AND i_a_id = a_id
ORDER BY i_title ASC LIMIT 50;
*************************** 1. row ***************************
  select_type: SIMPLE
        table: item
         type: index
possible_keys: i_i_a_id
          key: i_i_title
      key_len: 63
          ref: NULL
         rows: 50
        Extra: Using where
*************************** 2. row ***************************
  select_type: SIMPLE
        table: author
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 5
          ref: test.item.i_a_id
         rows: 1
        Extra:
```

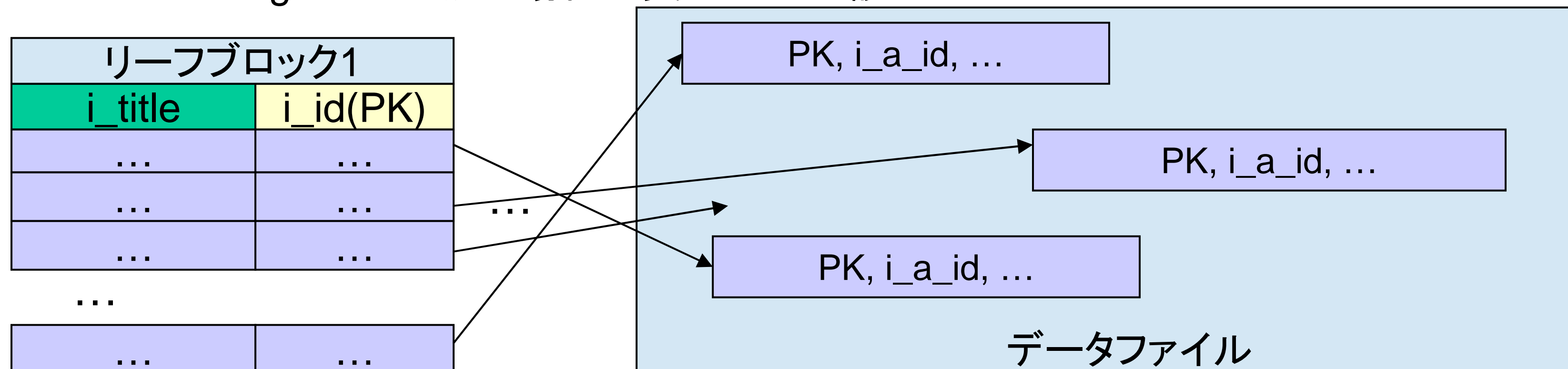
前提条件:
- itemには10000件、authorには2500件
- i_titleには単独のインデックス
- item, authorの順番でジョイン
- a_idはauthorテーブルの主キー
- LIKEの中間一致検索ではインデックスは使えない
- WHERE句にマッチするレコードはほとんど無い

Copyright 2008 Sun Microsystems inc

ケーススタディ:DBT-1 (2)

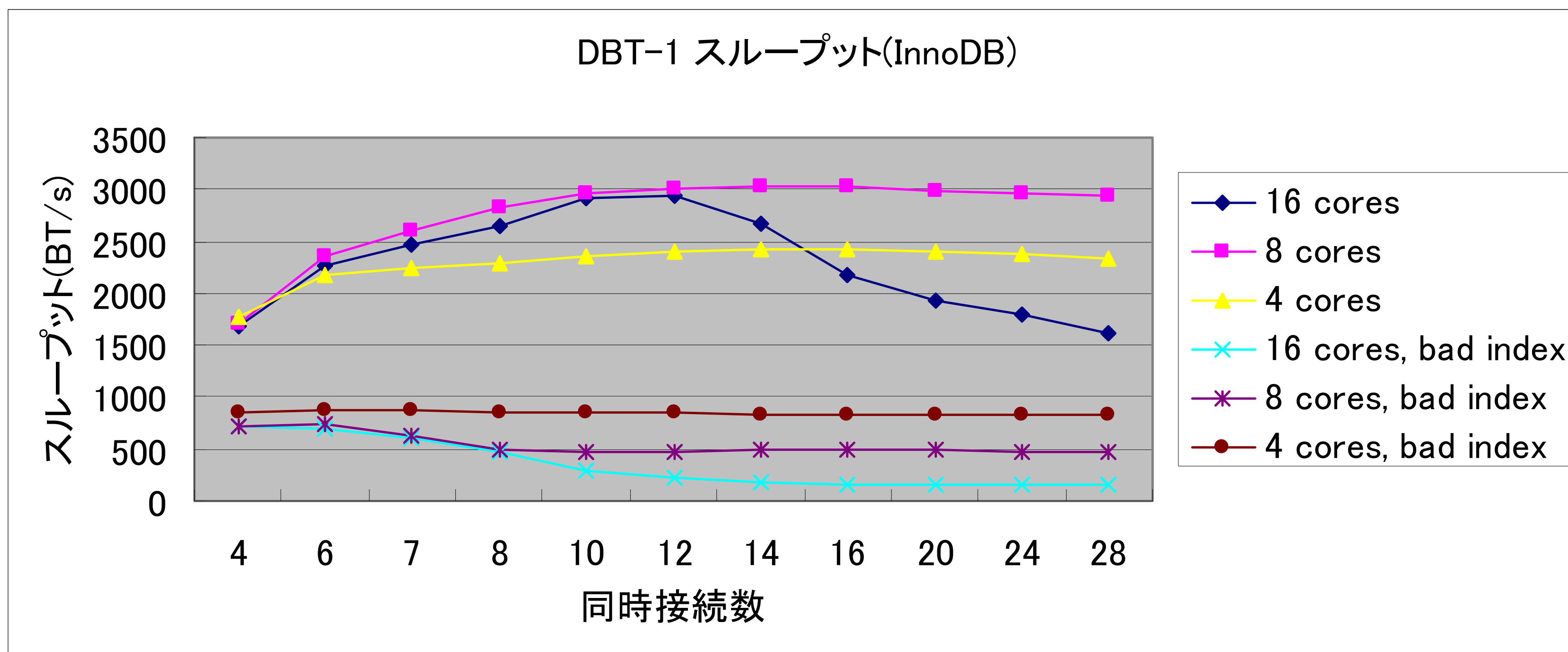
```
SELECT i_id, i_title, a_fname, a_lname FROM item, author
WHERE i_title LIKE '%BABABAOGBABAIN%' AND i_a_id = a_id
ORDER BY i_title ASC LIMIT 50;
```

Type=「index」でExtraに「Using index」が無いと何が起こるか
・type=indexはフルインデックススキャン
・Using indexが無い場合は実レコードも読む



・インデックスを先頭から順番に読んでいく
・対応するレコードをデータファイルから読む
・WHERE句の一致判定をする
・i_a_id列を用いてジョインし、対応するレコードがauthorテーブルにあるかどうか判定する
・マッチするレコードが50件になるか、すべてを読み終えるまで繰り返す
・DBT-1のデータの場合、条件を満たすレコードが5件しか無いので、インデックスを全件読まないといけない
・InnoDBの場合、セカンダリインデックスのスキャンはCPUスケーラビリティを悪化させる

ケーススタディ: DBT-1 (3)



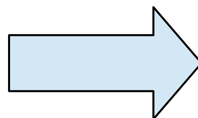
・下3本は、ORDER BY LIMIT Nで適切な実行計画が選ばれなかったケース。
IPAから発信されているベンチマーク結果は、これがベース
・上3本は、IGNORE INDEXヒントを使い、実行計画をコントロールしたケース
・実行計画が適切な場合は、8コアまでは普通にスケールしておりスループットは3000を超えている
・実行計画が不適切な場合は、8コアや16コアだと逆にスループットが落ちる。
最大の4コアでもスループットは3分の1以下。
・5.0.58, innodb_thread_concurrency=0

更新処理とインデックス

INSERTすると何が起こるのか

```
INSERT INTO tbl (key1) VALUES (61)
```

リーフブロック1	
key1	RowID
1	10000
2	5
3	15321
…	
60	431



リーフブロック1	
key1	RowID
1	10000
2	5
3	15321
…	
60	431

リーフブロック2	
key1	RowID
61	15322
空き状態	

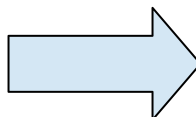
・リーフブロックが一杯で、
これ以上エントリが入らない

・新しいリーフブロックが割り当てられて、
その中に追加される

昇順INSERT

```
INSERT INTO tbl (key1) VALUES (now())
```

リーフブロック1	
key1	RowID
2008-08-01	10000
2008-08-02	5
2008-08-03	15321
…	
2008-10-29	431



リーフブロック1	
key1	RowID
2008-08-01	10000
2008-08-02	5
2008-08-03	15321
…	
2008-10-29	431

リーフブロック2	
key1	RowID
2008-10-29	431
空き状態	

・リーフブロックが一杯で、
これ以上エントリが入らない

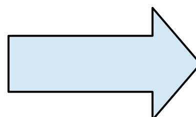
・新しいリーフブロックが割り当てられて、
その先頭から入る

・日付など、インデックスの並び順に対して昇順に入るインデックスもある
・新しいブロックは、空のブロックを新規に割り当てる
・1ブロックをフルに利用する
・虫食い状態になりにくい
・ブロック数が少なくなる
・サイズが小さい。ゆえにキャッシュされやすい

ランダムINSERT

INSERT INTO tbl (key1) VALUES (31)

リーフブロック1	
key1	RowID
1	10000
2	5
3	15321
…	
60	431



リーフブロック1	
key1	RowID
1	10000
…	
30	333
空き状態	

リーフブロック2	
key1	RowID
31	345
…	
60	431
空き状態	

・通常、インデックスの並び順に対してレコードはランダム順に入る
例： メッセージテーブルに対する、メンバーIDなど
・新しいブロックの中に、既存のブロックから半分移る

・虫食い状態になりやすい
・1ブロックあたりに占めるエントリ数が少なくなる傾向
・ブロック数が多くなる
・サイズが増える。ゆえにキャッシュされにくくなる

昇順INSERT vs ランダムINSERT

1000万件のレコードがすでに存在するテーブルに対して、
100万回追加INSERTするのにかかる時間と、インデックスサイズを測定

セカンダリインデックスの値を順番に入れていく (1000万1 .. 1100万)
vs
セカンダリインデックスの値をランダムに入れていく (1から1000万の間でランダム)

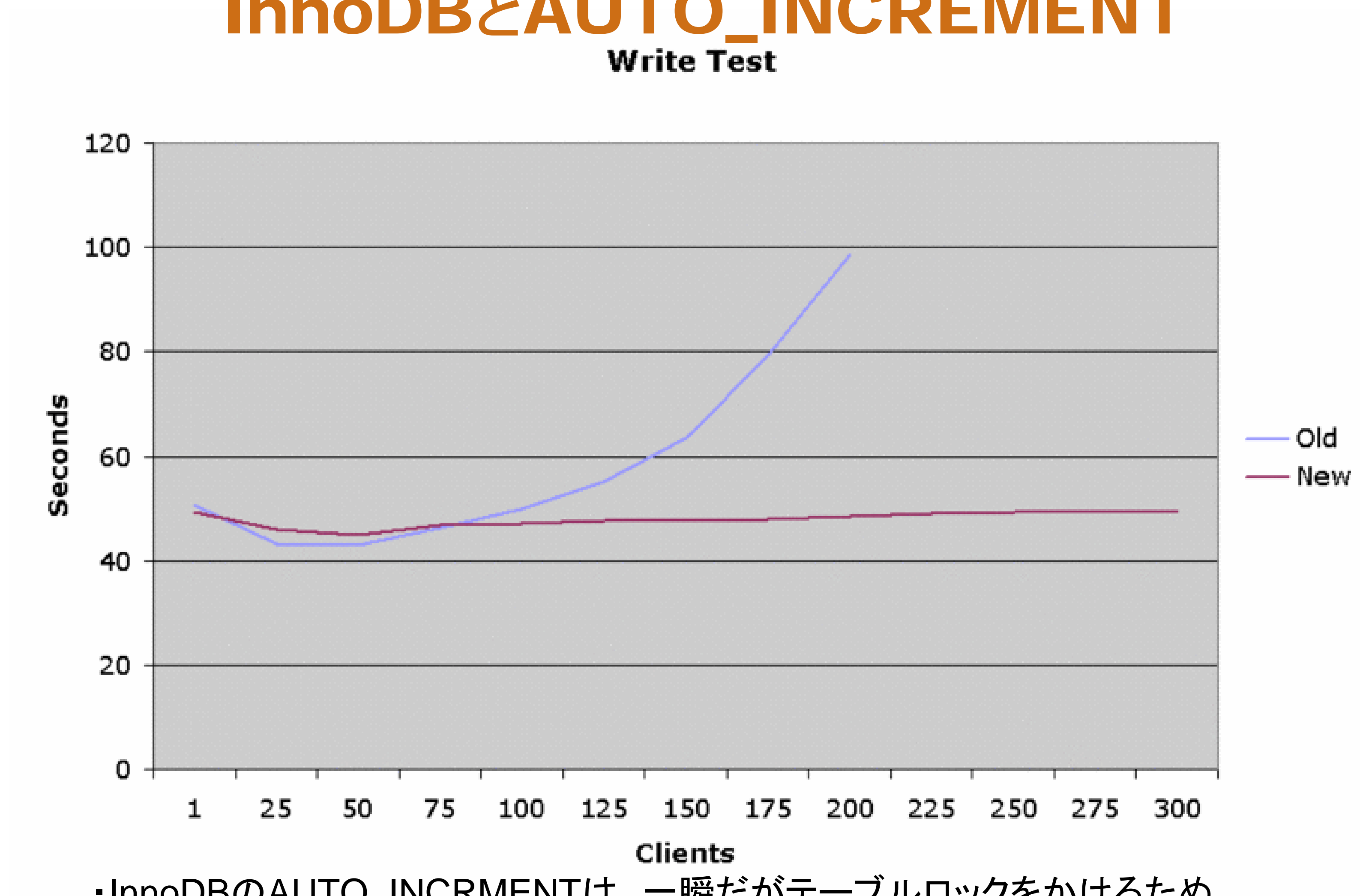
* どちらも、主キーはauto_increment
まったく同じことを、インデックス数が3個の場合でも実験

インデックス数1個	昇順INSERT	ランダムINSERT
100万件INSERT時間	37.88秒	56.72秒
インデックスサイズ	161MB	335MB

インデックス数3個	昇順INSERT	ランダムINSERT
100万件INSERT時間	52.26秒	3分4秒
インデックスサイズ	483MB	1.06GB

インデックスに対して昇順にINSERTすることがいかに重要かが分かる
InnoDBでは、特に主キーを昇順INSERTすることが重要

InnoDBとAUTO_INCREMENT



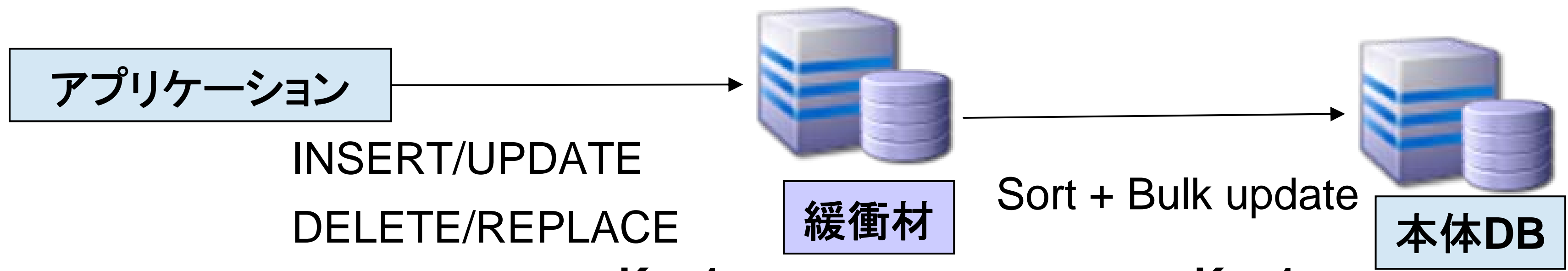
・InnoDBのAUTO_INCRMENTは、一瞬だがテーブルロックをかけるため、同時接続数の増加に対してスケーラビリティが劇的に落ちる

・5.1ではメカニズムを変えることで軽減

Copyright 2008 Sun Microsystems inc

昇順INSERTのためのアーキテクチャ

Buffering insert(緩衝材) パターン

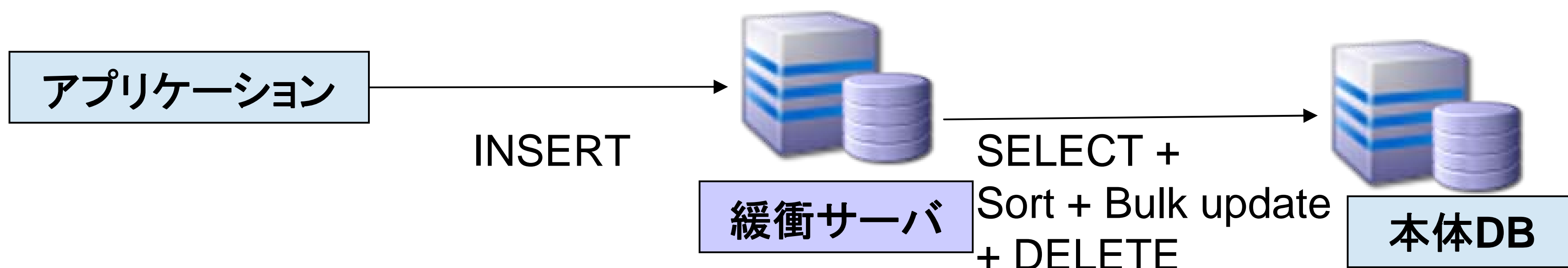


Key1	Key1
100	5
5	100
7932	998
998	1000
3325	3325
1000	7932

- **Web/Appサーバは緩衝材に対して更新して終了**
- **緩衝材から本体DBに対して更新**
  - **非同期**
  - **データを主要インデックスに対して昇順に並べ替える**
  - **バルクINSERT/UPDATE/DELETE**

緩衝材の候補:
- **インデックスの無いテーブル**
- **キャッシュサーバ**
- **キュー**

緩衝材:インデックスの無いテーブル

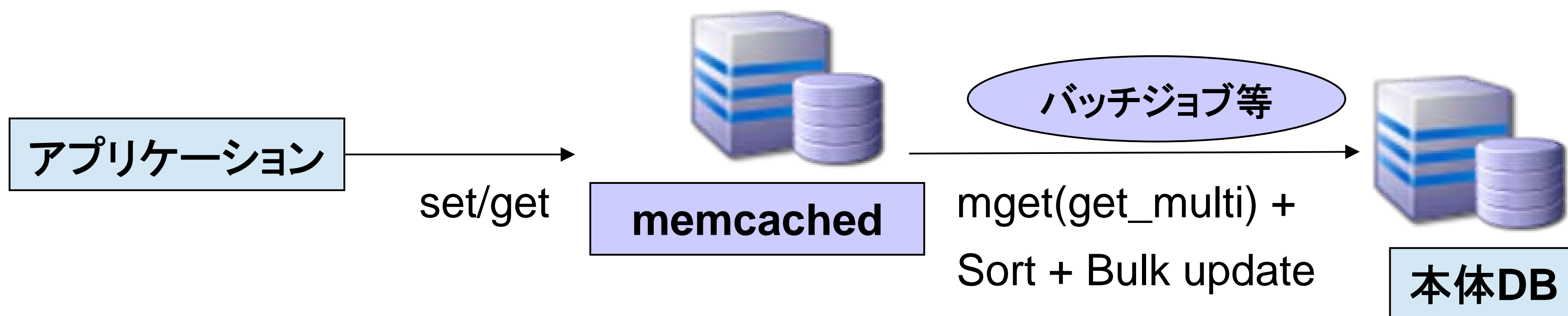


Key1		Key1
100		5
5	→	100
7932		998
998		1000
3325		3325
1000		7932

・本体DBと同じ列定義で、インデックスが無いテーブルを緩衝サーバ上に用意
・時間/日付ごとに緩衝サーバ上のテーブル名を変えていく
・古いテーブルの中身を本体DBに移す
・更新結果を即時に検索したい場合には向かない

Copyright 2008 Sun Microsystems inc

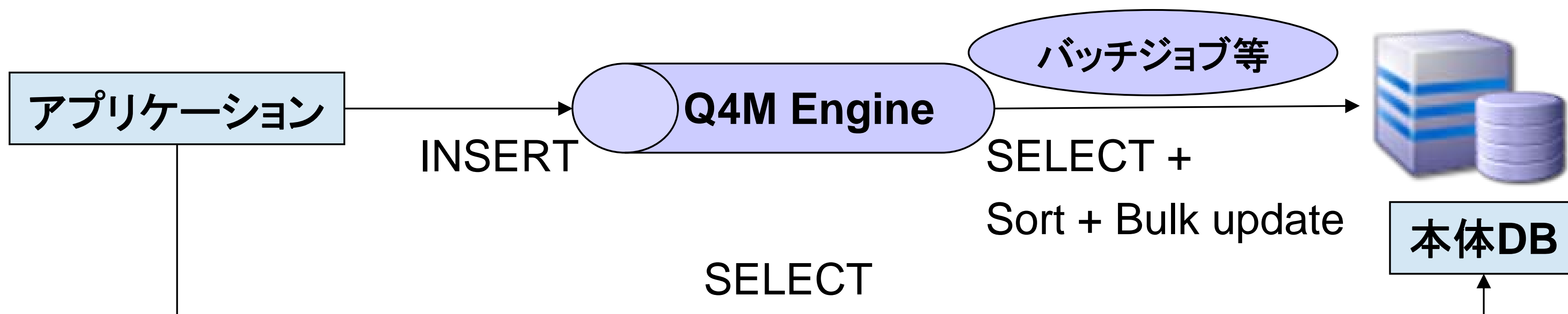
緩衝材：キャッシュサーバ (memcached)



- アプリケーションからはmemcachedにストアして終了(高速)
- バッチジョブなどが、後でまとめて値を本体DBに投入
  (複数件をまとめて取り、ソートしてバルクupdate)
- 検索処理は本体DBまたはmemcachedに向ける

- memcachedはクラッシュすると中身が失われるので注意が必要
- C/Perl/PHP/Ruby/Javaなどメジャーな言語用にライブラリが提供されている
- MySQL Enterprise購入ユーザには、memcachedも追加でサポート

緩衝材:キュー (Q4M Storage Engine)



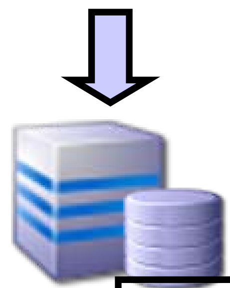
- **Q4M (Queue For MySQL: http://q4m.31tools.com)**
- サイボウズラボ 奥 一穂氏が開発した、**MySQL**のカスタムストレージエンジン
- **MySQL5.1**で利用可能 (**MySQL**本体の再ビルドは不要)
- クラッシュしてもデータは失われない
- 「**mixi**エコー」や「**Pathtraq**」など本番環境での実績がある
- バッチジョブなどを作成して、本体**DB**に反映

- 検索処理は本体**DB**に向かう
- 本体への反映の遅延を考慮して、**memcached**などに別途最新データを置いておく手も有効

Copyright 2008 Sun Microsystems inc

参考: Blackhole Storage Engine

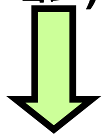
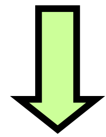
アプリケーションからの更新



CREATE TABLE tbl1 … ENGINE=Blackhole;
INSERT INTO tbl1 VALUES(1);
→バイナリログには通常通り記録。
テーブルの中身は空のまま

マスター

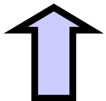
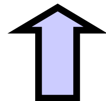
(レプリケーション)

・マスターの更新負荷を軽減する上で効果的なストレージエンジン
・メリット：アプリケーション側の変更がいらない
・デメリット：発行されたSQL文がそのまま実行されるだけなので、インデックス昇順INSERTの恩恵を受けることはできない

スレーブ

スレーブ

アプリケーションからの参照

・スレーブでは、ENGINE=InnoDBなどでテーブルを作っておく
・バイナリログ経由でINSERTが実行される
→InnoDBなど通常のテーブルに格納

Bad Practice – インデックスの数が多すぎる

- インデックスの数が多すぎる
  - カラムごとにインデックス
  - 1テーブルに20個のインデックス
  - アクセスパターンを全部網羅

- インデックスが過度に多いと
  - 更新性能が落ちる
  - 消費サイズが増える
  - インデックスの利用効率が落ちると、キャッシュ効率が悪くなり検索性能も落ちる

- 必要なもの「だけ」にインデックスを作る

Bad Practice – インデックスが大きすぎる

- 例
  - URL (20～100バイト級)
  - UUID (36バイト)

- 現象
  - スペースを消費し、パフォーマンス悪化
  - 投入順序と並び順が一致していなければ、断片化が大きく発生する

- 対処策
  - Prefix Index機能を使い、先頭Nバイトだけをインデックス化する(例：INDEX col(5))
  - 代替となるインデックスを別に用意する
    - CRC32()による4バイトハッシュ値をインデックスに使うと、サイズを小さくできる

url varchar(255) index(url)

SELECT xx FROM tbl WHERE url='http://…….' ;

→

url_hash integer unsigned, index(url_hash)

**SELECT xx FROM tbl WHERE url='http://……..'
AND url_hash= CRC32('http://………');**

Copyright 2008 Sun Microsystems inc

Bad Practice – データ型の不正比較

- SELECT xx FROM t1 WHERE varchar_column = 1;
  - varchar_columnにインデックスがあっても、使われない
  - varchar_column = “1”と指定しなければならない
  - varchar_column = 1の場合、”001”,”01”, “1.00”なども条件を満たす
  - 文字列型は、文字順に並ぶ
    - 001
    - 002
    - 003
    - 01
    - 02
    - 1
    - 1.00
    - 2
  - この条件でインデックスを使うことはできない

- データ型を合わせるのは定石の1つ

Bad Practice – 無意味なマルチカラムインデックス

- マルチカラムインデックスの1列目を指定していない
  - SELECT xx FROM t1 WHERE key_part2 = xx;
- 1列目で範囲検索、2列目以降でイコール検索
  - SELECT xx FROM t1 WHERE key_part1 between 100 AND 105 AND key_part2 = ‘abcde’ ;
- 「Using index」に落とし込むために、意図的に余分な列をインデックスに入れることがある
  - SELECT id FROM tbl WHERE c1 = xx AND (c2 = xx or c3 = xx) で、c1,c2,c3をマルチカラムインデックスに

Bad Practice – ハッシュインデックスで範囲検索(できない)

- create table working_table (c1 int , c2 int, index(c1)) engine=memory;
- 100万件くらい投入
- 1万件ずつSELECTを100回繰り返すことを意図して、
- select * from working_table where c1 between 1 and 10000;
- フルテーブルスキャンになる

- MEMORYは、デフォルトで「ハッシュインデックス」を作り、「Btreeインデックス」を作らない
- ハッシュインデックスはイコール検索しかできず、範囲検索(<, >, BETWEEN, LIKEなど)はできない

- create table working_table (c1 int , index using btree(c1)) engine=memory;

Bad Practice – データ準備の欠如

- テストデータが
  - 10ユーザしかない
  - 100万ユーザいるけど、10ユーザしかテストでは使わない
  - 全ユーザが同じ商品を注文することになっている
- データ量が少ないと全部キャッシュされるので、
  読み取り時にディスクアクセスが発生しない
- データ量が多くても、アクセス範囲が狭ければ全部 キャッシュされる
- データに偏りがあるとインデックス戦略が変わる
  - ORDER BY LIMITなど
- 不適切なインデックスがあっても、それに気づきにくい
- 本番のアクセスパターンと全く違うテストは
  負荷テストの意味がまったく無い
- せめて量と偏りだけは考えておく

まとめ

- ランダムアクセス回数を最小化することが
インデックス検索性能を高める鍵
- EXPLAINを使い、実行計画を注視する
  - 範囲検索は、Covering Indexへの帰結を狙う
  - 必要に応じ、FORCE/IGNORE INDEXによってコントロールする
- データ型、データサイズなど基本を守る

- 更新性能を抜本的に高めるためには、
昇順INSERTが有効