MySQLのロックについて

JPOUG> SET EVENTS 20140907

2014/09/07 平塚 貞夫

Revision 2

自己紹介

- DBエンジニアをやっています。専門はOracle DatabaseとMySQL。
  - オープンソースソフトウェアの導入支援をしています。
  - 仕事の割合はOracle：MySQL：PostgreSQL＝1：2：7くらいです。

- Twitter：@sh2nd
- はてな：sh2
- 

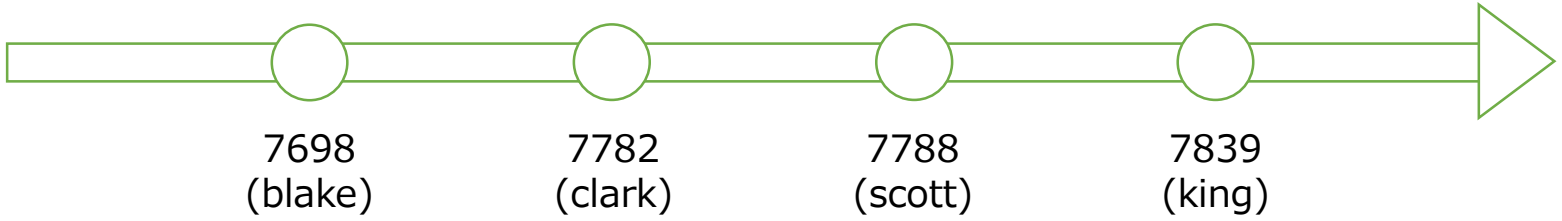
- 写真は実家で飼っているミニチュアダックスのオス、アトムです。

本日のお題

想定外のデッドロック

- MySQLのInnoDBストレージエンジンに対して、2つのトランザクションを以下の順番で実行するとデッドロックが発生します。

```
1:REPEATABLE_READ
 2:REPEATABLE_READ
1:UPDATE:DELETE FROM emp WHERE empno = 7784
(1:UPDATE:COUNT=0)
 2:UPDATE:DELETE FROM emp WHERE empno = 7786
 (2:UPDATE:COUNT=0)
1:UPDATE:INSERT INTO emp (empno, ename) VALUES (7784, 'steve')
 2:UPDATE:INSERT INTO emp (empno, ename) VALUES (7786, 'bill')
(1:UPDATE:COUNT=1)
 (2:UPDATE:COUNT=0)
 (2:com.mysql.jdbc.exceptions.jdbc4.MySQLTransactionRollbackException:
  Deadlock found when trying to get lock; try restarting transaction)
 2:ABORT
```



- このデッドロックの発生メカニズムを理解するために、InnoDBのロックアーキテクチャについて確認していきます。

従業員テーブル

- emp(従業員)テーブルを用いて確認していきます。
  empno(社員番号)、ename(社員名)、job(職種),mgr(上司の社員番号)、
  hiredate(入社日)、sal(給料)、comm(歩合給)、deptno(部門番号)

empno	ename	job	mgr	hiredate	sal	comm	deptno
7369	smith	clerk	7902	1980-12-17	800.00	NULL	20
7499	allen	salesman	7698	1981-02-20	1600.00	300.00	30
7521	ward	salesman	7698	1981-02-22	1250.00	500.00	30
7566	jones	manager	7839	1981-04-02	2975.00	NULL	20
7654	martin	salesman	7698	1981-09-28	1250.00	1400.00	30
7698	blake	manager	7839	1981-05-01	2850.00	NULL	30
7782	clark	manager	7839	1981-06-09	2450.00	NULL	10
7788	scott	analyst	7566	1987-04-19	3000.00	NULL	20
7839	king	president	NULL	1981-11-17	5000.00	NULL	10
7844	turner	salesman	7698	1981-09-08	1500.00	0.00	30
7876	adams	clerk	7788	1987-05-23	1100.00	NULL	20
7900	james	clerk	7698	1981-12-03	950.00	NULL	30
7902	ford	analyst	7566	1981-12-03	3000.00	NULL	20
7934	miller	clerk	7782	1982-01-23	1300.00	NULL	10

- empnoカラムにプライマリインデックス、jobカラムに非ユニークインデックスを作成してあります。

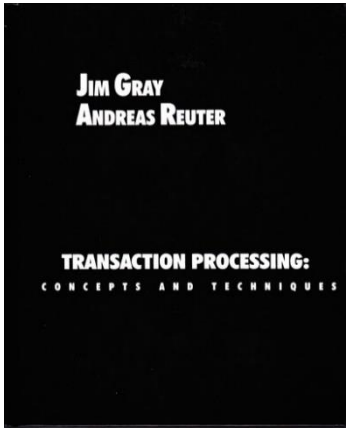
InnoDBの設計思想

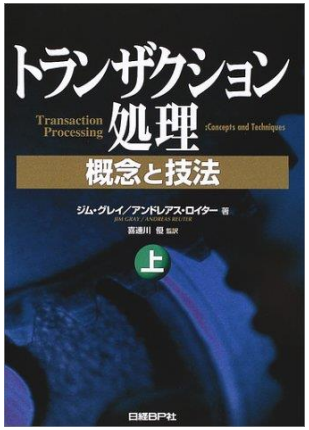
トランザクション処理 概念と技法

- InnoDBは「トランザクション処理 概念と技法」という本の内容を参考にして実装されています。

  "The book by Jim Gray and Andreas Reuter, Transaction processing, from about year 1992, is the best reference. Much of InnoDB has been written according to the instructions in that book. - Heikki Tuuri"
  (http://lists.mysql.com/mysql/107555)

- 洋書は購入可能です。
- 和書は中古で手に入れるしかないと思います。





ファントムリード

- ファントムリードとは、他のトランザクションによってINSERTされたレコードが自分のトランザクションで見えてしまう現象のことです。

- READ COMMITTEDの場合にファントムリードが発生する例です。

```
1:READ_COMMITTED
 2:READ_COMMITTED
1:QUERY:SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 ORDER BY empno
(empno      ename      job        mgr        hiredate   sal        comm       deptno     )
(7782       clark      manager    7839       1981-06-09 2452.00    null       10         )
(7788       scott      analyst    7566       1987-04-19 3002.00    null       20         )
(1:QUERY)
 2:UPDATE:INSERT INTO emp (empno, ename) VALUES (7785, 'steve')
 (2:UPDATE:COUNT=1)
 2:COMMIT
1:QUERY:SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 ORDER BY empno
(empno      ename      job        mgr        hiredate   sal        comm       deptno     )
(7782       clark      manager    7839       1981-06-09 2452.00    null       10         )
(7785       steve      null       null       null       null       null       null       )
(7788       scott      analyst    7566       1987-04-19 3002.00    null       20         )
(1:QUERY)
```

ファントムリードを防ぐ

- トランザクション分離レベルの定義によれば、ファントムリードはSERIALIZABLE以外のトランザクション分離レベルで発生する可能性があります。

分離レベル	ダーティリード	ファジーリード	ファントムリード
READ UNCOMMITTED	可能性あり	可能性あり	可能性あり
READ COMMITTED	発生しない	可能性あり	可能性あり
REPEATABLE READ (InnoDBのデフォルト)	発生しない	発生しない	可能性あり (InnoDBでは発生しない)
SERIALIZABLE	発生しない	発生しない	発生しない

- 本書の第7章でトランザクション分離性における課題としてファントムリードが挙げられており、それに対処するために述語ロック、粒度ロック、キー範囲ロック、後方キーロックや前方キーロックといった技法が紹介されています。

- 多くのDBMSがデフォルトのトランザクション分離レベルをREAD COMMITTEDに設定していますが、本書の内容を参考にして実装されたInnoDBは、より高いトランザクション分離性を実現することを目指しているように見受けられます。

ファントムリードを防ぐ例 その1

- InnoDBでトランザクション分離レベルがREPEATABLE READの場合は、ファントムリードを防ぐことが可能です。

- なおREPEATABLE READはファントムリードが発生することを許容していますが、DBMS側の都合により発生しない場合でも定義上の問題はありません。

```
1:REPEATABLE READ
 2:READ_COMMITTED
1:QUERY:SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 ORDER BY empno
(empno      ename      job        mgr        hiredate   sal        comm       deptno     )
(7782       clark      manager    7839       1981-06-09 2452.00    null       10         )
(7788       scott      analyst    7566       1987-04-19 3002.00    null       20         )
(1:QUERY)
 2:UPDATE:INSERT INTO emp (empno, ename) VALUES (7785, 'steve')
 (2:UPDATE:COUNT=1)
 2:COMMIT
1:QUERY:SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 ORDER BY empno
(empno      ename      job        mgr        hiredate   sal        comm       deptno     )
(7782       clark      manager    7839       1981-06-09 2452.00    null       10         )
(7788       scott      analyst    7566       1987-04-19 3002.00    null       20         )
(1:QUERY)
```

ファントムリードを防ぐ例 その2

- SELECT文にLOCK IN SHARE MODE、あるいはFOR UPDATE句を付与して明示的にロックを取得することができます。これをロッキングリードと呼びます。

- REPEATABLE READでロッキングリードを行うことによっても、ファントムリードを防ぐことが可能です。この場合は他のトランザクションによるINSERTが待たされます。

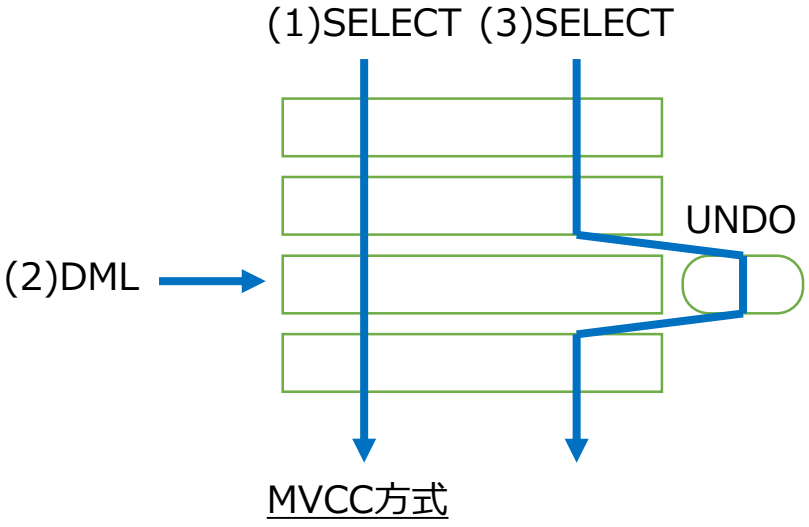
```
1:REPEATABLE_READ
 2:READ_COMMITTED
1:QUERY:SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 ORDER BY empno LOCK IN SHARE MODE
(empno      ename      job        mgr        hiredate   sal        comm       deptno     )
(7782       clark      manager    7839       1981-06-09 2452.00    null       10         )
(7788       scott      analyst    7566       1987-04-19 3002.00    null       20         )
(1:QUERY)
 2:UPDATE:INSERT INTO emp (empno, ename) VALUES (7785, 'steve')
 (2:UPDATE:COUNT=0)
 (2:java.sql.SQLException: Lock wait timeout exceeded; try restarting transaction)
 2:ABORT
1:QUERY:SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 ORDER BY empno
(empno      ename      job        mgr        hiredate   sal        comm       deptno     )
(7782       clark      manager    7839       1981-06-09 2452.00    null       10         )
(7788       scott      analyst    7566       1987-04-19 3002.00    null       20         )
(1:QUERY)
```

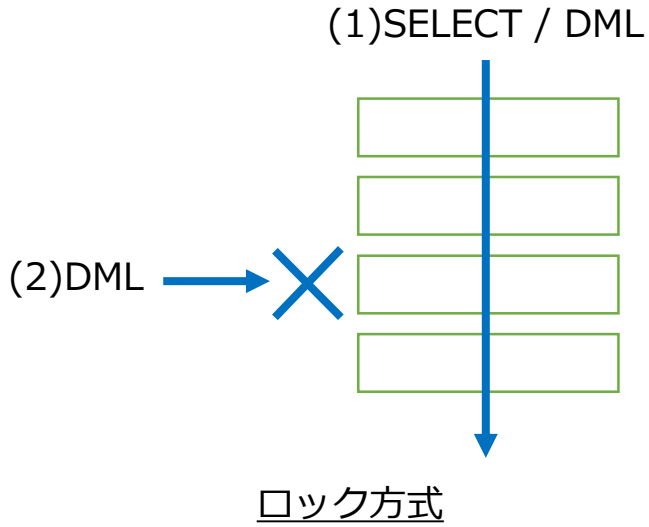
MVCC方式とロック方式

- トランザクション分離性を実現する方式として、MVCC(Multi-Version Concurrency Control)方式とロック方式があります。

- MVCC方式ではSELECT文が共有ロックを取得せず、他のトランザクションによる更新があった場合はUNDOを用いて更新前のデータを見せます。

- ロック方式では先行するSQL文が共有ロックまたは排他ロックを取得し、他のトランザクションが排他ロックを取得して行う更新を待たせます。





MVCC方式とロック方式の使い分け

- InnoDBがMVCC方式とロック方式をどのように使い分けているのかを以下に示します。REPEATABLE READ以上でネクストキーロックというものを取得するところ、それからSERIALIZABLEでSELECT文が共有ロックを取得するところが特徴です。

分離レベル	SELECT	LOCK IN SHARE MODE	FOR UPDATE / DML
READ COMMITTED	文単位のMVCC	共有レコードロック	排他レコードロック
REPEATABLE READ	トランザクション単位のMVCC	共有ネクストキーロック	排他ネクストキーロック
SERIALIZABLE	共有ネクストキーロック	共有ネクストキーロック	排他ネクストキーロック

- Oracle Databaseの場合は以下のようになります。

分離レベル	SELECT	LOCK IN SHARE MODE	FOR UPDATE / DML
READ COMMITTED	文単位のMVCC	(なし)	排他レコードロック
(SET TRANSACTION READ ONLY)	トランザクション単位のMVCC	(なし)	(禁止)
SERIALIZABLE	トランザクション単位のMVCC	(なし)	排他レコードロック (シリアル化エラーあり)

ここまでのまとめ

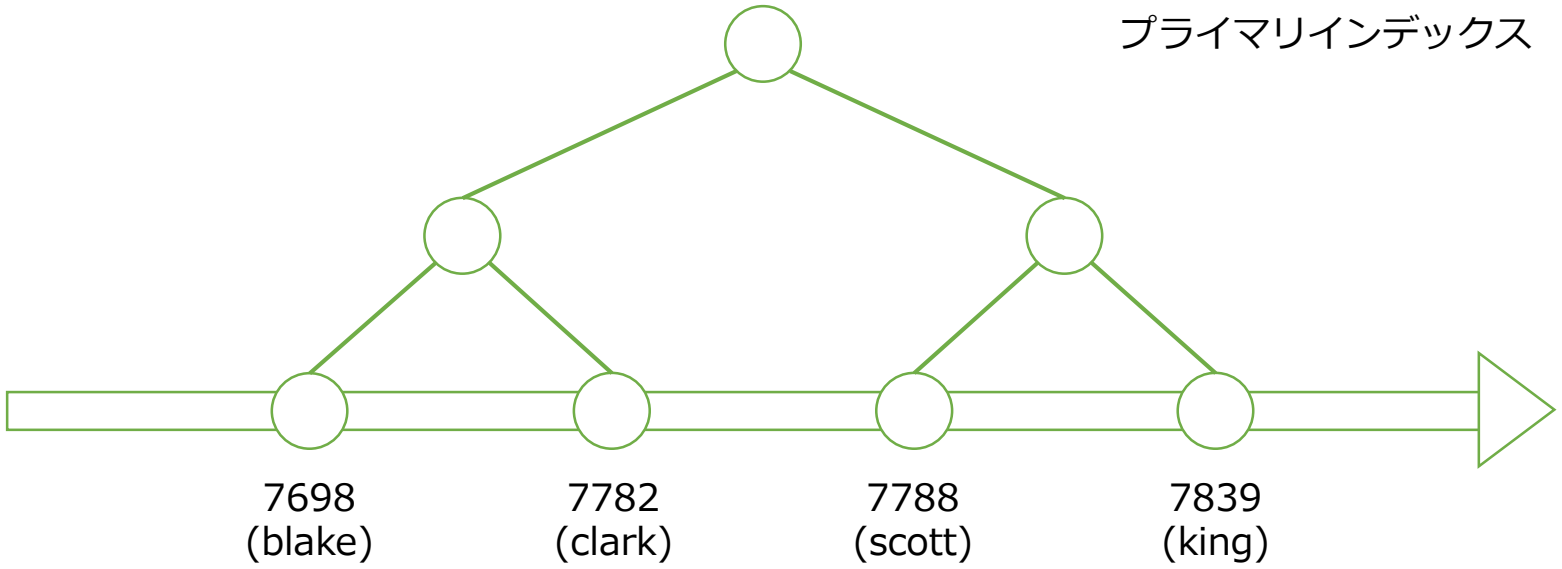
- InnoDBは「トランザクション処理 概念と技法」という本の内容を参考にして実装されており、ファントムリードを防ぐことを目指しているように見受けられます。

- ファントムリードを防ぐためには、トランザクション単位のMVCCを実行するか、あるいはネクストキーロックというものを取得します。

InnoDBのロックアーキテクチャ

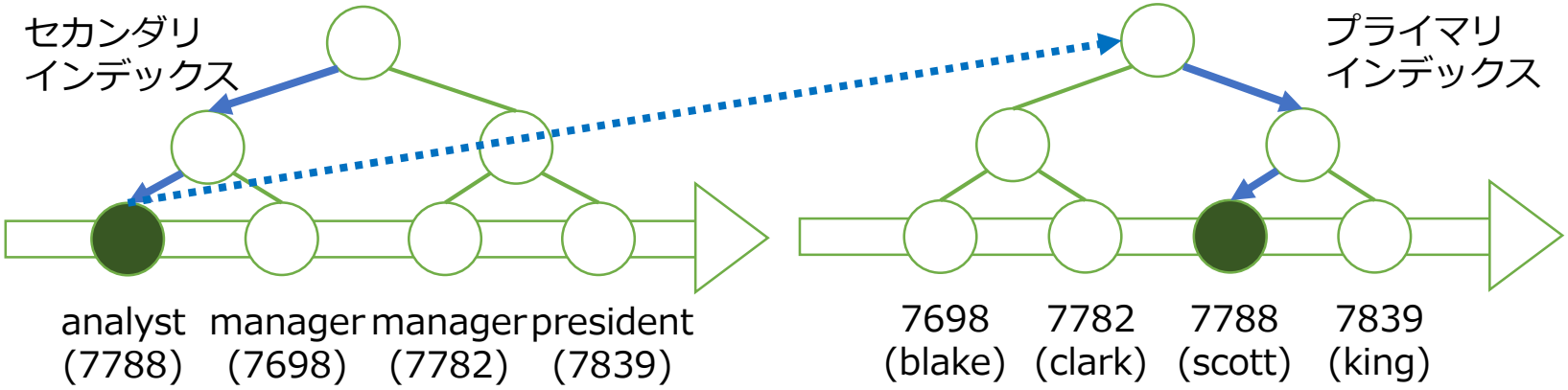
テーブルの構造

- InnoDBは、プライマリインデックスのリーフノードにデータを格納するクラスタインデックス構造を採用しています。Oracle Databaseで言う索引構成表です。



インデックスの構造

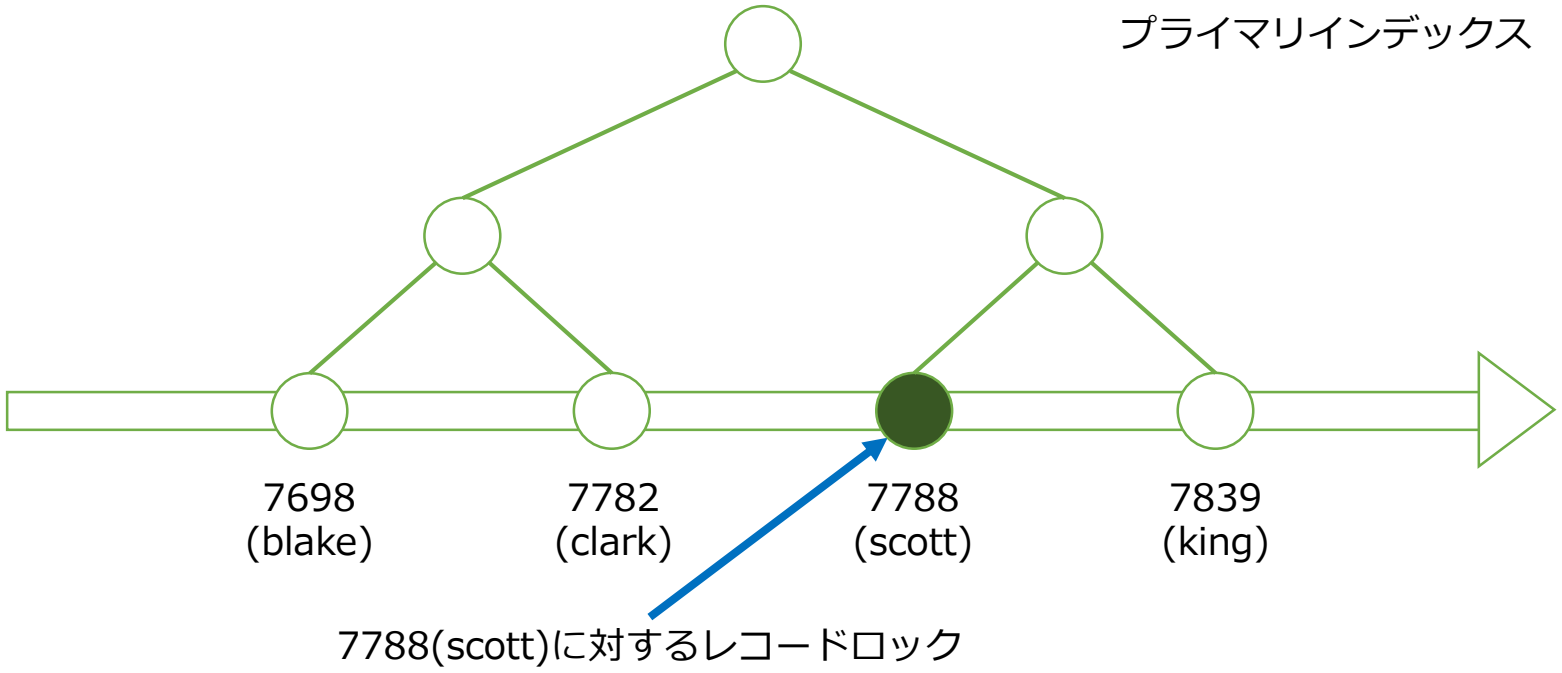
- プライマリインデックス以外のインデックスのことを、セカンダリインデックスと呼びます。セカンダリインデックスはリーフノードにプライマリキーの値を格納しており、セカンダリインデックスを走査したあとプライマリインデックスを走査することで目的のレコードを取得します。



- Oracle Databaseの場合はインデックスのリーフノードにレコードの物理的な位置を示すROWIDを格納しており、インデックスを2回走査することはありません。

レコードロック

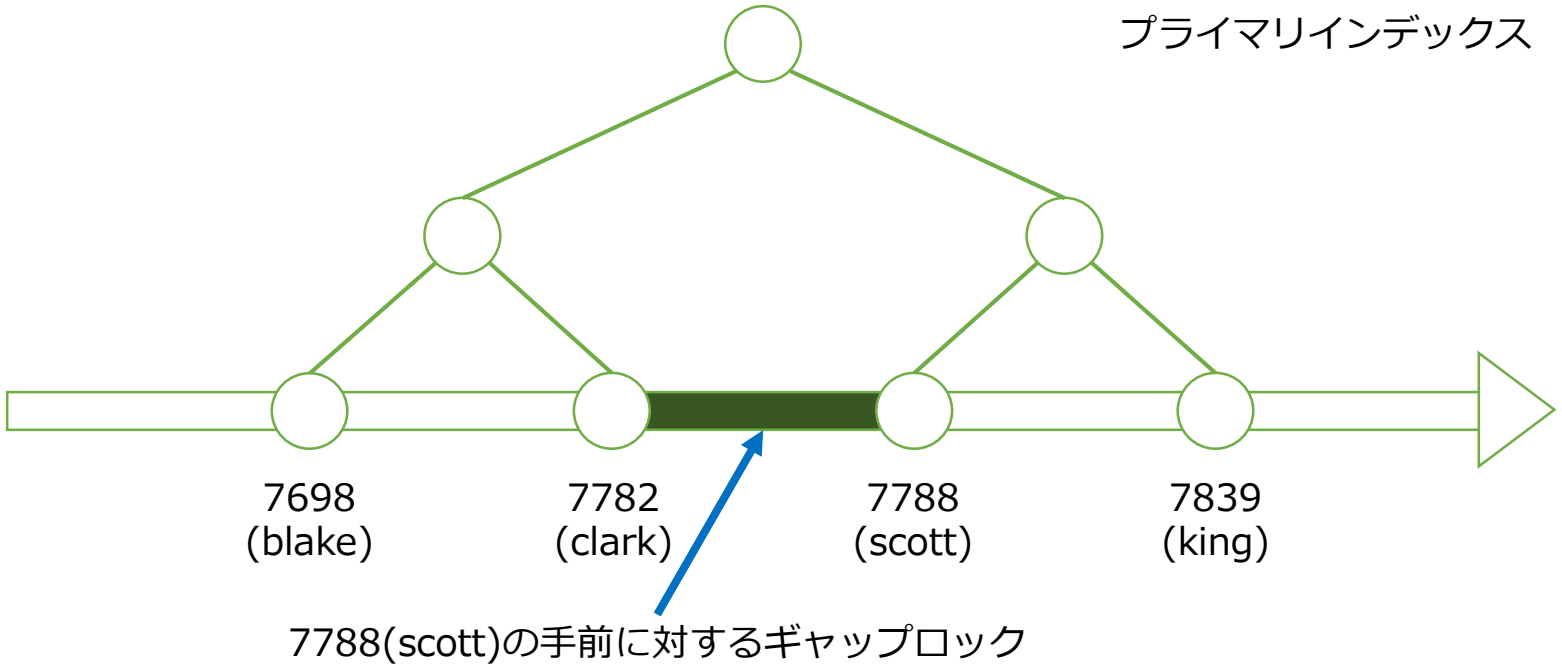
- InnoDBはインデックス上のレコードをロックすることでレコードロックを行います。



ギャップロック

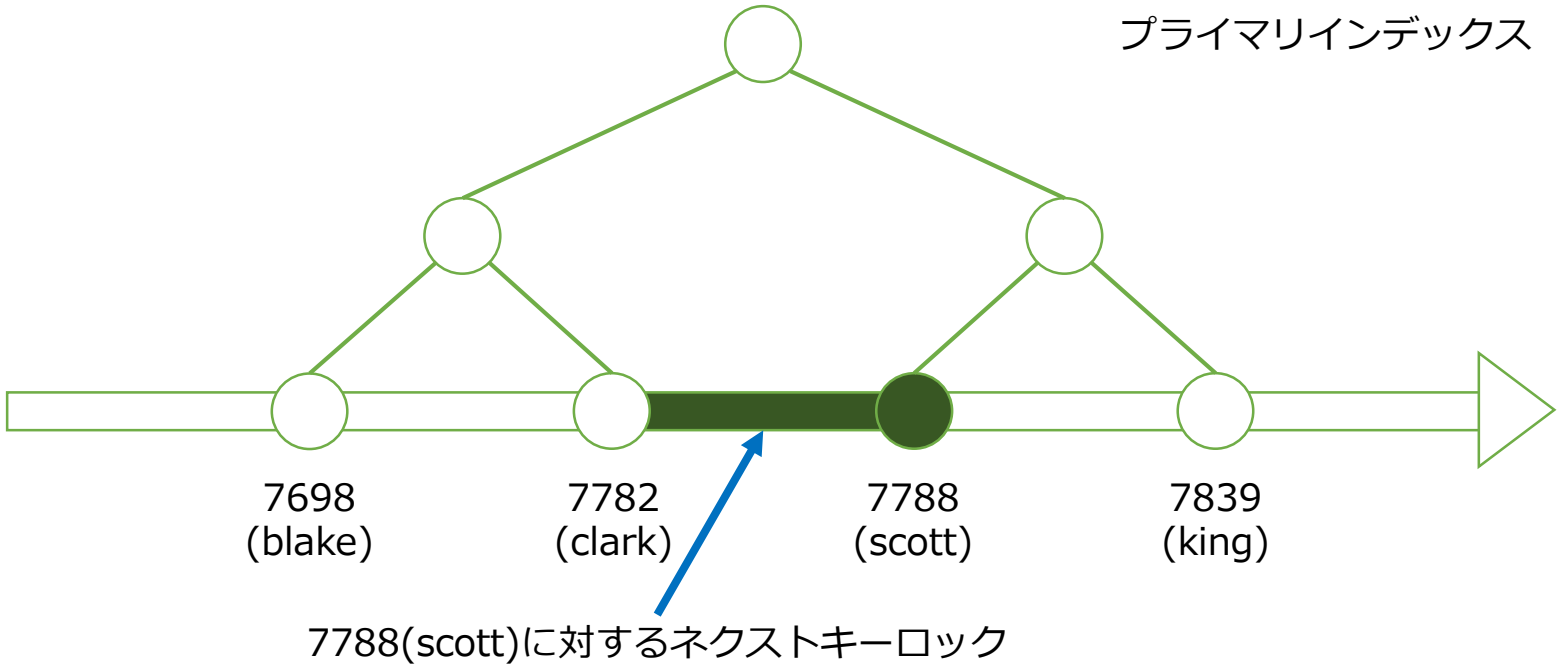
- InnoDBではインデックス上のレコードとレコードの間がロックされることがあります。これをギャップロックと呼びます。

- ロック方式でファントムリードを防ぐには、現時点で存在しないレコードをロックする必要があります。ギャップロックは存在しないレコードを低コストでロックするために導入された仕組みです。



ネクストキーロック

- レコードロックとその手前のギャップロックを合わせて、ネクストキーロックと呼びます。



ロックの範囲

- SELECT LOCK IN SHARE MODE、SELECT FOR UPDATEやDMLを実行すると、InnoDBはインデックス上で走査したレコードに対してレコードロック、ギャップロックまたはネクストキーロックを取得します。

- 検索条件に合致したレコードに対してではなく、走査したレコードに対してロックを取得するというところが特徴です。InnoDBはあくまでMySQLのストレージエンジンであり、検索時にすべての検索条件を把握できているわけではないことが要因の一つだと考えられます。

ここまでのまとめ

- InnoDBはクラスタインデックス構造を採用しています。
- インデックス上のレコードをロックすることでレコードロックを行います。
- ファントムリードを防ぐために、レコードとレコードの間をロックする仕組みが導入されています。
- インデックス上で走査したレコードに対してロックを取得します。

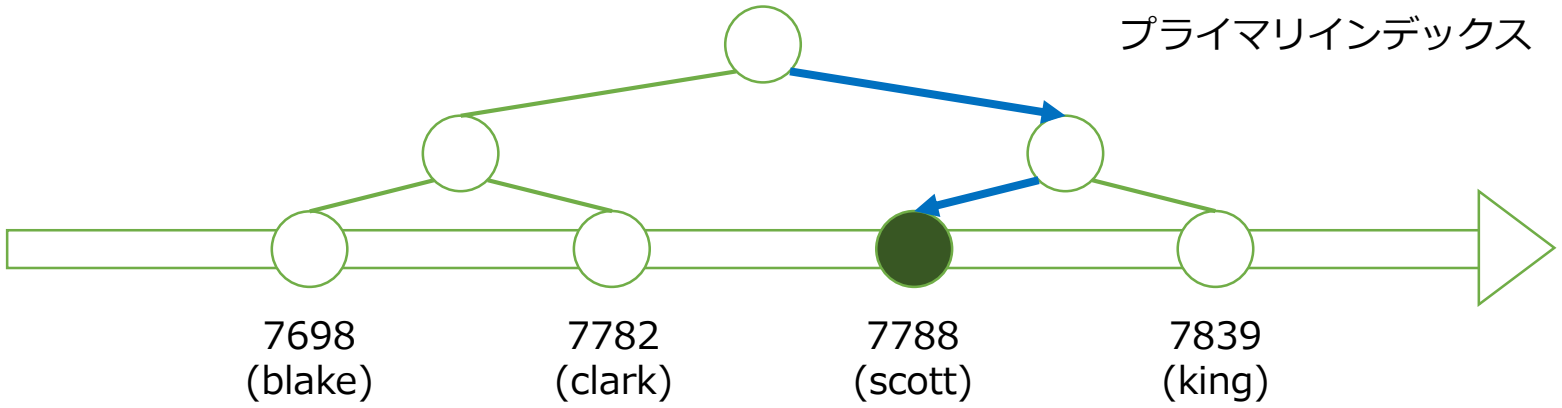
REPEATABLE READにおけるロック範囲

プライマリインデックスに対する等価検索

```
SELECT * FROM emp WHERE empno = 7788 FOR UPDATE
```

- 以下のロックが取得されます。
  - 7788(scott)に対するレコードロック

- ファントムリードが発生する余地はないため、ギャップロックは取得されません。

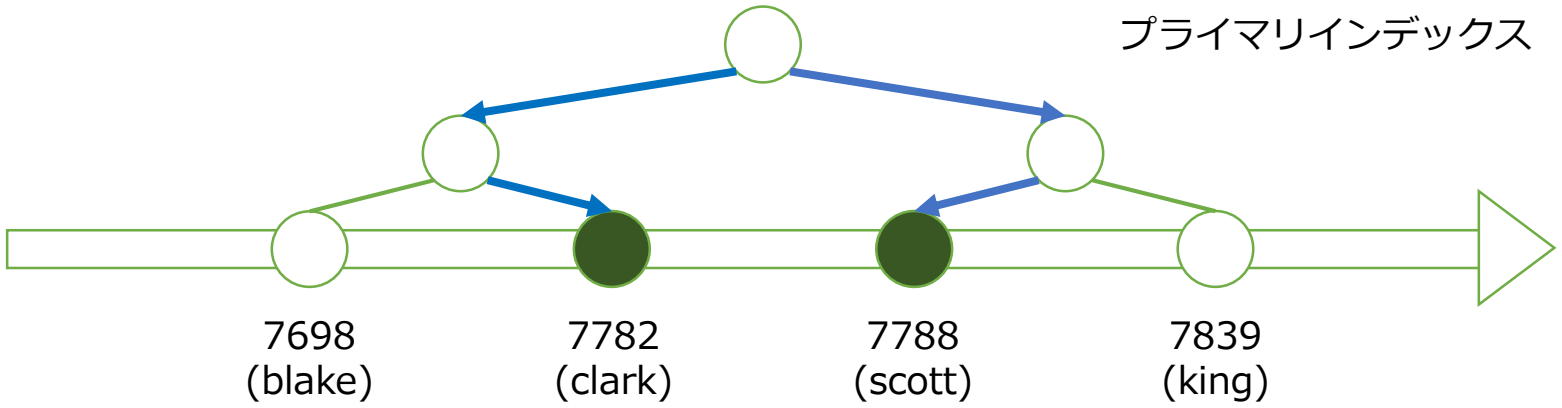


プライマリインデックスに対するIN検索

```
SELECT * FROM emp WHERE empno IN (7782, 7788) FOR UPDATE
```

- 以下のロックが取得されます。
  - 7782(clark)、7788(scott)に対するレコードロック

- IN検索は、等価検索を複数回実行することと同じです。

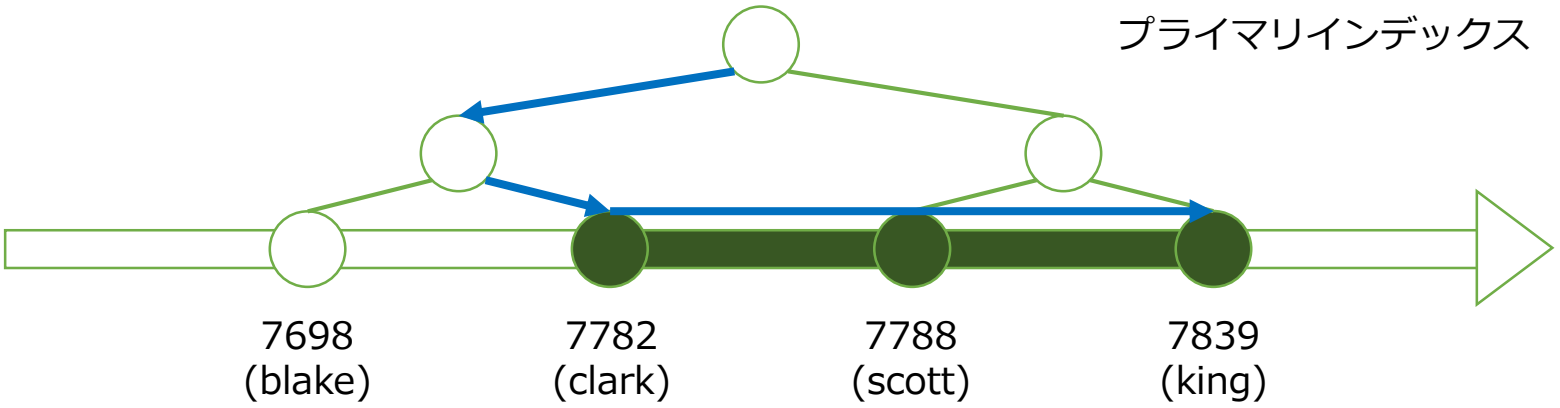


プライマリインデックスに対する範囲検索

```
SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 FOR UPDATE
```

- 以下のロックが取得されます。
  - 7782(clark)に対するレコードロック
  - 7788(scott)、7839(king)に対するネクストキーロック

- 範囲検索の場合はリーフノードのリスト構造を走査し、7839(king)まで走査して止まります。そのため一つ先の7839(king)に対してもネクストキーロックが取得されます。ロックの範囲が広くなることに注意が必要です。

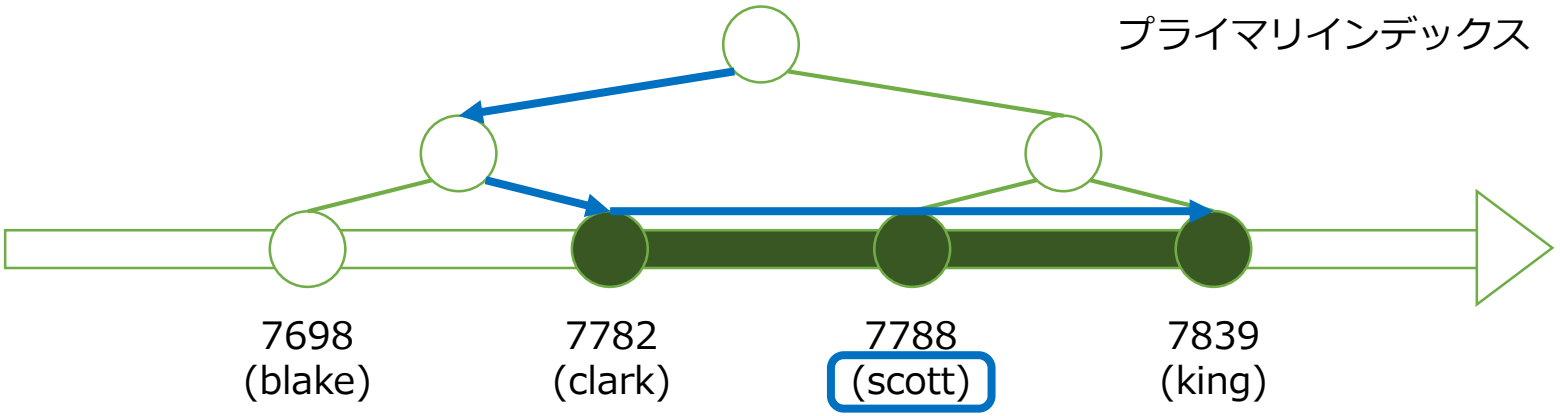


プライマリインデックスに対する範囲検索＋追加条件

```
SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 AND ename LIKE '%t' FOR UPDATE
```

- 以下のロックが取得されます。
  - 7782(clark)に対するレコードロック
  - 7788(scott)、7839(king)に対するネクストキーロック

- 7782(clark)と7839(king)は追加条件に合致しませんが、ロックは取得されたままとなります。

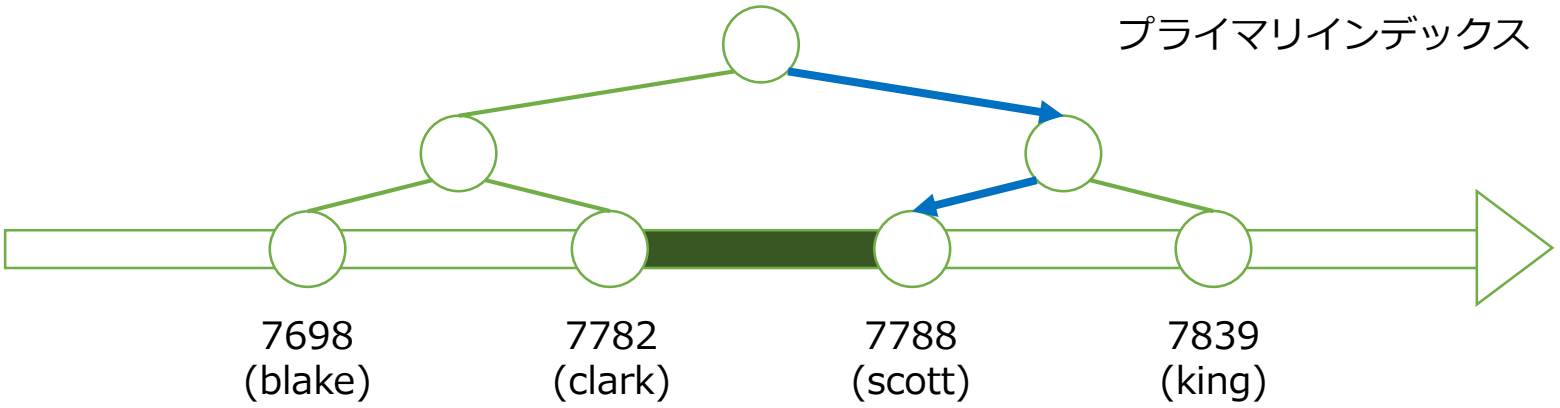


プライマリインデックスに対する等価検索、空振り

```
SELECT * FROM emp WHERE empno = 7785 FOR UPDATE
```

- 以下のロックが取得されます。
  - 7788(scott)の手前に対するギャップロック

- 検索条件に合致した場合はレコードロックが取得されますが、空振りした場合はギャップロックが取得されます。ロックの範囲が広くなることに注意が必要です。

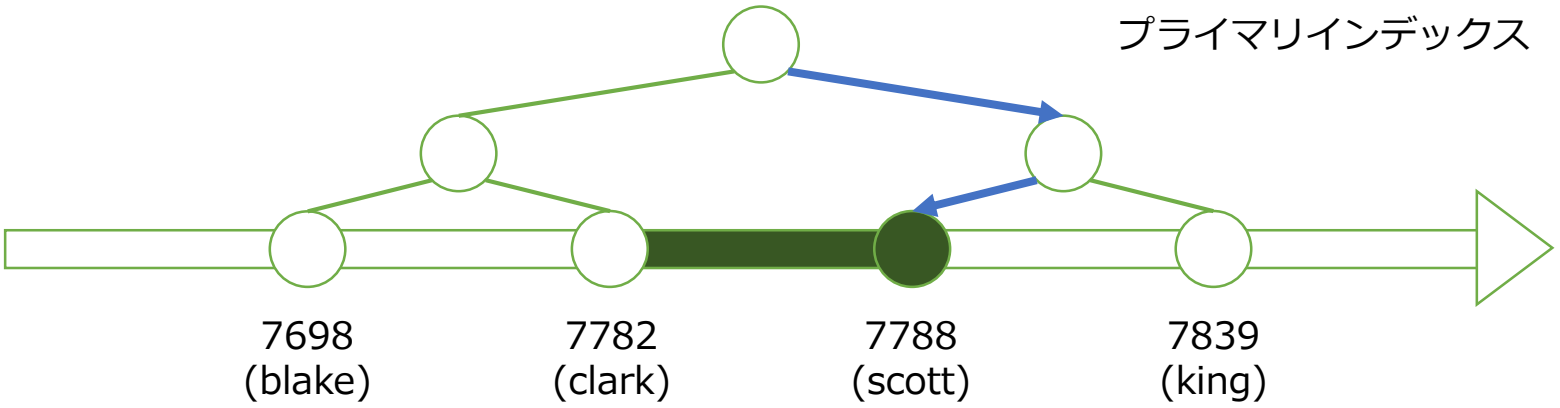


プライマリインデックスに対する範囲検索、空振り

```
SELECT * FROM emp WHERE empno BETWEEN 7784 AND 7786 FOR UPDATE
```

- 以下のロックが取得されます。
  - 7788(scott)に対するネクストキーロック

- 範囲検索ではリーフノードのリスト構造を走査しますが、この場合は7788(scott)だけを走査して止まります。7788(scott)に対してネクストキーロックが取得されます。

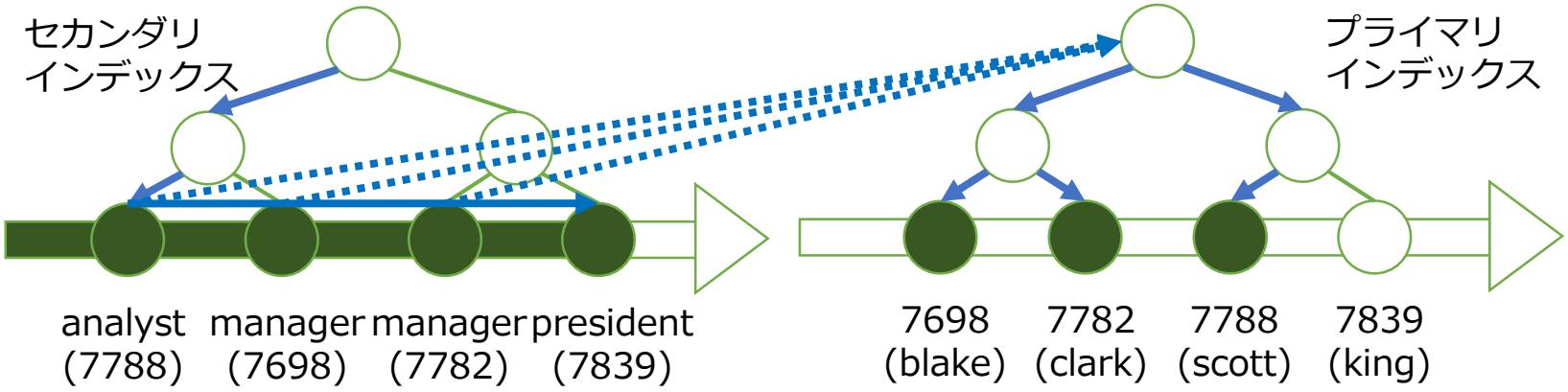


非ユニークインデックスに対する範囲条件

```
SELECT * FROM emp WHERE job BETWEEN 'analyst' AND 'manager' FOR UPDATE
```

- セカンダリインデックスに対する以下のロックが取得されます。
  - analyst(7788)、manager(7698)、manager(7782)、president(7839)に対するネクストキーロック
- プライマリインデックスに対する以下のロックが取得されます。
  - 7698(blake)、7782(clark)、7788(scott)に対するレコードロック

- それぞれのインデックスに対してロックが取得されます。

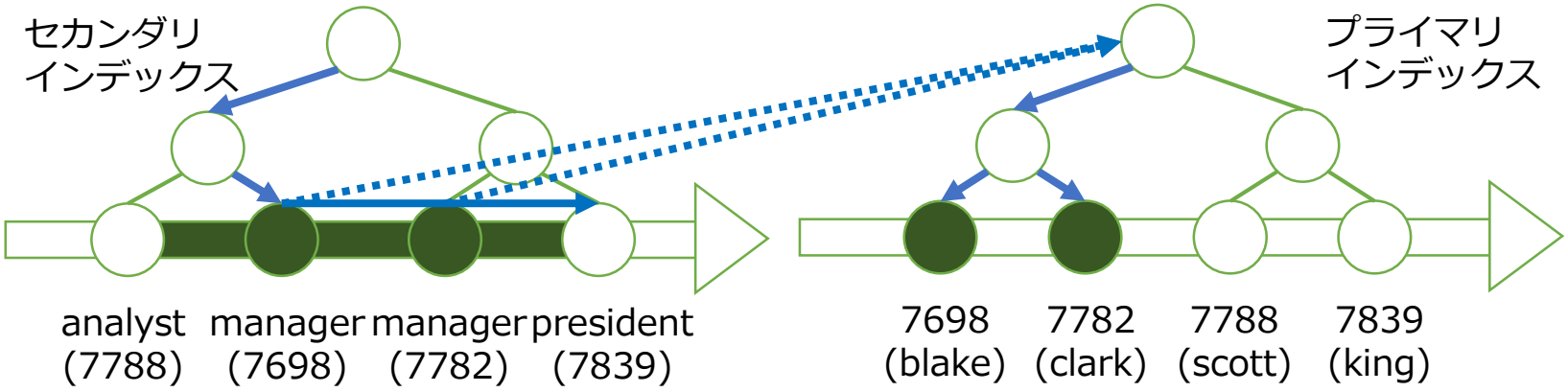


非ユニークインデックスに対する等価条件

```
SELECT * FROM emp WHERE job = 'manager' FOR UPDATE
```

- セカンダリインデックスに対する以下のロックが取得されます。
  - manager(7698)、manager(7782)に対するネクストキーロック
  - president(7839)の手前に対するギャップロック
- プライマリインデックスに対する以下のロックが取得されます。
  - 7698(blake)、7782(clark)に対するレコードロック

- 非ユニークインデックスの場合は、等価条件であっても範囲条件に近い挙動となります。また、一つ先のpresident(7839)に対してはネクストキーロックではなくギャップロックが取得されます。



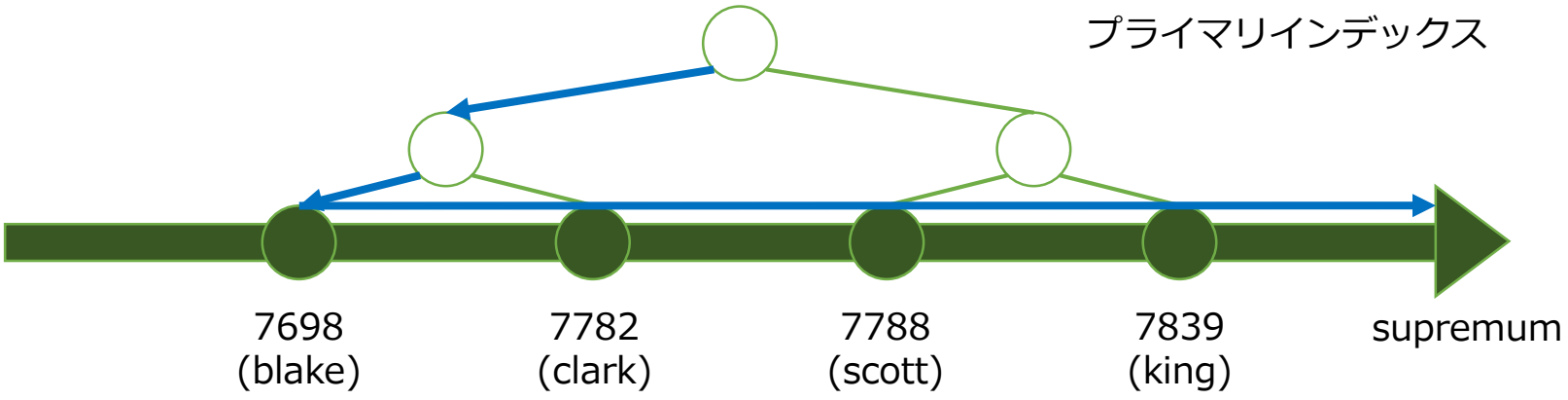
非ユニークインデックスに対する等価条件、フルスキャン

```
SELECT * FROM emp IGNORE INDEX (emp_job) WHERE job = 'manager' FOR UPDATE
```

- プライマリインデックスに対する以下のロックが取得されます。
  - 7698(blake)、7782(clark)、7788(scott)、7839(king)、supremumに対するネクストキーロック

- InnoDBは走査したレコードに対してロックを取得するため、SQL実行計画が変化するとロックの範囲も変化します。

- supremumとは、内部的に設けられている上限値のレコードです。



ここまでのまとめ

- プライマリインデックスに対する等価検索の場合は、レコードロックが取得されます。
- プライマリインデックスに対する範囲検索の場合は、走査したレコードに対するレコードロックまたはネクストキーロックが取得されます。また、一つ先のレコードまで走査します。
- 検索が空振りした場合は、ギャップロックまたはネクストキーロックが取得されます。
- 非ユニークインデックスの場合は、セカンダリインデックスとプライマリインデックスのそれぞれに対してロックが取得されます。また、等価条件であっても範囲条件に近い挙動となります。
- SQL実行計画が変化するとロックの範囲も変化します。

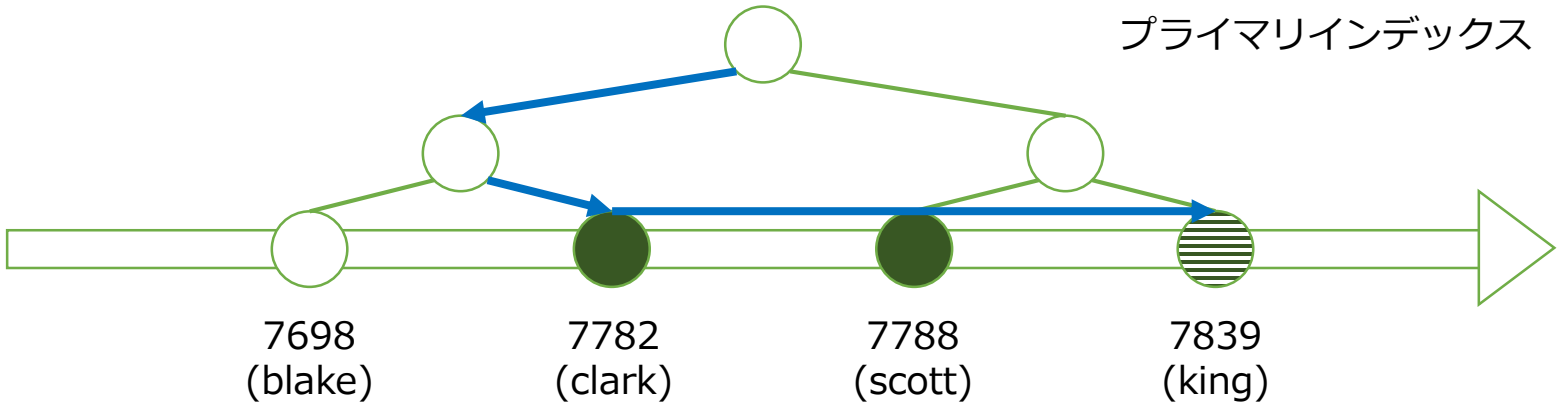
READ COMMITTEDにおけるロック範囲

プライマリインデックスに対する範囲検索

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 FOR UPDATE
```

- 以下のロックが取得されます。
  - 7782(clark)、7788(scott)、7839(king)に対するレコードロック
- SQL文の完了時に以下のロックが解放されます。
  - 7839(king)に対するレコードロック

- READ COMMITTEDはファントムリードを許容するため、ギャップロックは取得されません。また、検索条件に合致しなかったレコードに対するロックはSQL文の完了時に解放されます。

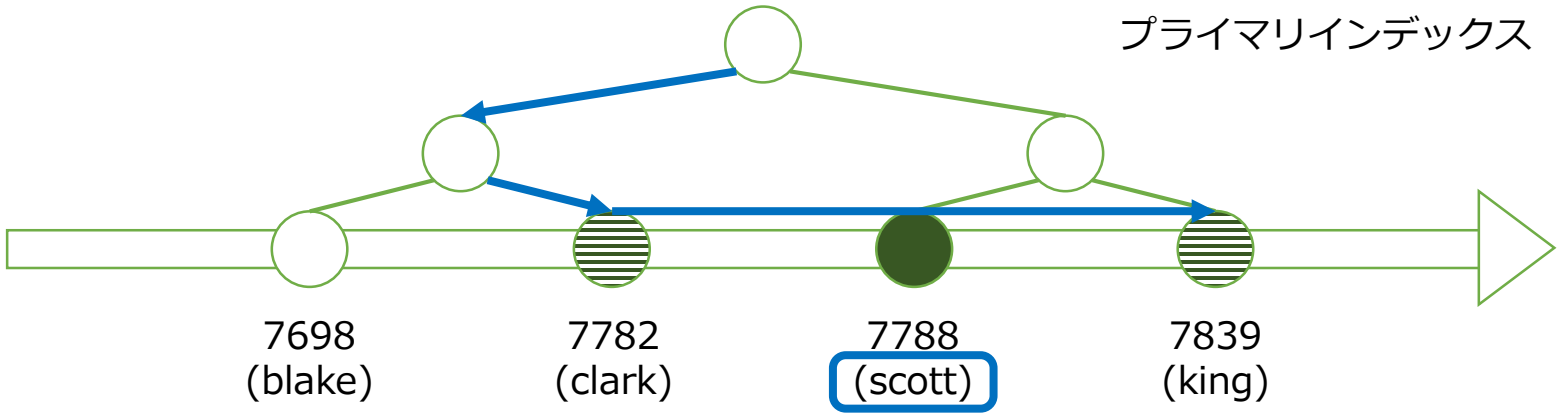


プライマリインデックスに対する範囲検索＋追加条件

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 AND ename LIKE '%t' FOR UPDATE
```

- 以下のロックが取得されます。
  - 7782(clark)、7788(scott)、7839(king)に対するレコードロック
- SQL文の完了時に以下のロックが解放されます。
  - 7782(clark)、7839(king)に対するレコードロック

- インデックスが関与しない追加条件についても、合致しなかったレコードに対するロックはSQL文の完了時に解放されます。



トランザクションの順番による挙動の違い

- 一時的に7782(clark)、7839(king)に対するレコードロックを取得するため、トランザクションの順番によって挙動が変わります。
- 後続トランザクションは7782(clark)のロックを取得することが可能です。

```
1:READ_COMMITTED
 2:READ_COMMITTED
1:QUERY:SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 AND ename LIKE '%t' FOR UPDATE
(empno      ename      job        mgr        hiredate   sal        comm       deptno     )
(7788       scott      analyst    7566       1987-04-19 3000.00    null       20         )
(1:QUERY)
 2:QUERY:SELECT * FROM emp WHERE empno = 7782 FOR UPDATE
 (empno      ename      job        mgr        hiredate   sal        comm       deptno     )
 (7782       clark      manager    7839       1981-06-09 2450.00    null       10         )
 (2:QUERY)
```

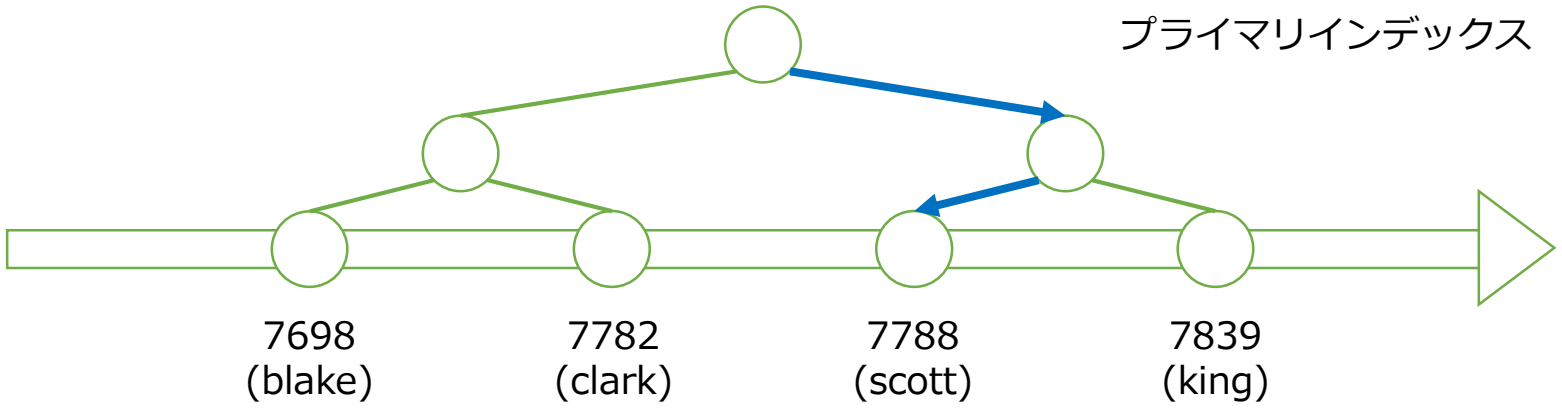
- 一方、7782(clark)のロックを取得している先行トランザクションがあると待たされます。

```
 2:READ_COMMITTED
1:READ_COMMITTED
 2:QUERY:SELECT * FROM emp WHERE empno = 7782 FOR UPDATE
 (empno      ename      job        mgr        hiredate   sal        comm       deptno     )
 (7782       clark      manager    7839       1981-06-09 2450.00    null       10         )
 (2:QUERY)
1:QUERY:SELECT * FROM emp WHERE empno BETWEEN 7782 AND 7788 AND ename LIKE '%t' FOR UPDATE
(1:QUERY)
(1:java.sql.SQLException: Lock wait timeout exceeded; try restarting transaction)
1:ABORT
```

プライマリインデックスに対する等価検索、空振り

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * FROM emp WHERE empno = 7785 FOR UPDATE
```

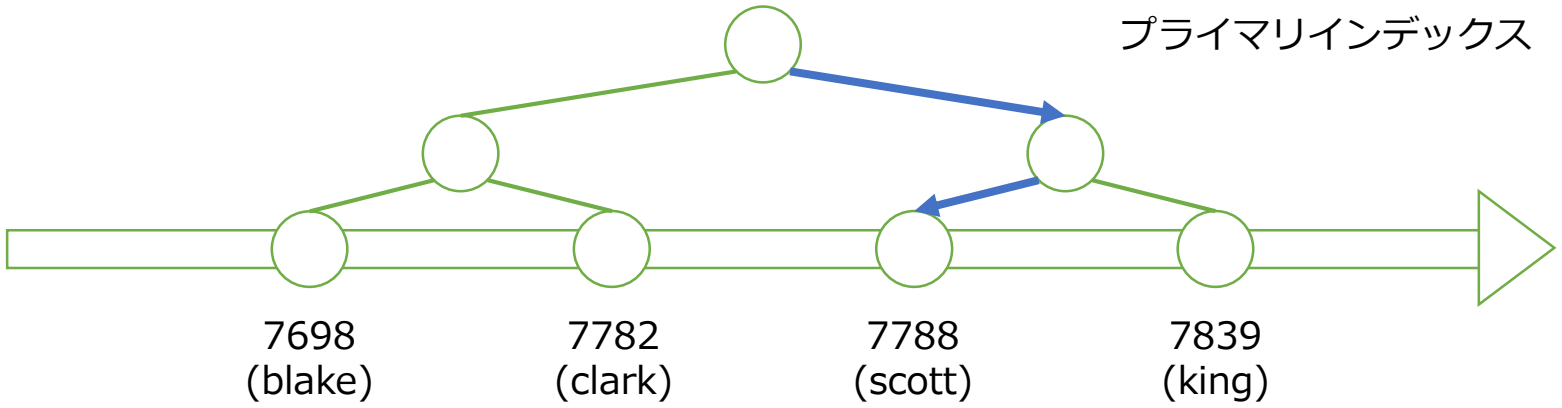
- ロックは取得されません。



プライマリインデックスに対する範囲検索、空振り

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * FROM emp WHERE empno BETWEEN 7784 AND 7786 FOR UPDATE
```

- ロックは取得されません。



ここまでのまとめ

- READ COMMITTEDの場合は、ギャップロックは取得されません。

- 走査したレコードに対するレコードロックが取得されますが、検索条件に合致しなかったレコードに対するロックはSQL文の完了時に解放されます。

- 検索が空振りした場合は、ロックは取得されません。

調査方法

InnoDBロックモニタ

- MySQL 5.6.16以降でパラメータinnodb_status_output_locksを有効にすると、SHOW ENGINE INNODB STATUSコマンドでロックの状態を確認できます。この機能のことをInnoDBロックモニタと呼びます。

```
SET GLOBAL innodb_status_output_locks = ON
SHOW ENGINE INNODB STATUS
```

- 実行例を以下に示します。

```
SELECT * FROM emp WHERE empno BETWEEN 7698 AND 7782 OR empno = 7835 FOR UPDATE ← 7835は空振り
```

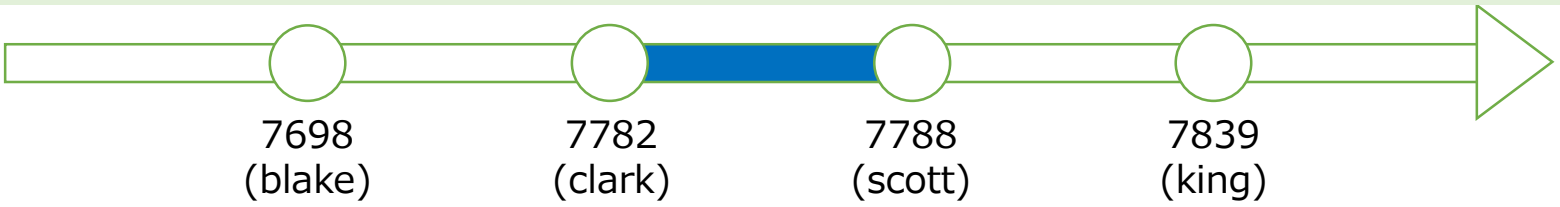
```
RECORD LOCKS space id 342 page no 3 n bits 88 index `PRIMARY` of table `scott`.`emp` trx id 1857327
lock_mode X locks rec but not gap   ← 7698(blake)に対するレコードロック
Record lock, heap no 7 PHYSICAL RECORD: n_fields 10; compact format; info bits 0
 0: len 4; hex 80001e12; asc     ;; ← 10進数で7698

RECORD LOCKS space id 342 page no 3 n bits 88 index `PRIMARY` of table `scott`.`emp` trx id 1857327
lock_mode X                         ← 7782(clark)と7788(scott)に対するネクストキーロック
Record lock, heap no 8 PHYSICAL RECORD: n_fields 10; compact format; info bits 0
 0: len 4; hex 80001e66; asc    f;; ← 10進数で7782
Record lock, heap no 9 PHYSICAL RECORD: n_fields 10; compact format; info bits 0
 0: len 4; hex 80001e6c; asc    l;; ← 10進数で7788

RECORD LOCKS space id 342 page no 3 n bits 88 index `PRIMARY` of table `scott`.`emp` trx id 1857327
lock_mode X locks gap before rec    ← 7839(king)の手前に対するギャップロック
Record lock, heap no 10 PHYSICAL RECORD: n_fields 10; compact format; info bits 0
 0: len 4; hex 80001e9f; asc     ;; ← 10進数で7839
```

本日のお題(再)

デッドロックの発生メカニズム

1. TX1のDELETE文が空振りし、7788(scott)の手前のギャップロックを取得します。

```
1:UPDATE:DELETE FROM emp WHERE empno = 7784
```



2. TX2のDELETE文が空振りし、7788(scott)の手前のギャップロックを取得します。なおギャップロック同士は競合しません。

```
2:UPDATE:DELETE FROM emp WHERE empno = 7786
```

3. TX1のINSERT文が7788(scott)の手前のギャップに対して挿入インテンションギャップロックの取得を試み、TX2のギャップロックと競合します。挿入インテンションギャップロックとは、INSERT文の実行時に取得される特殊なギャップロックです。挿入インテンションギャップロック同士は競合せず、通常のギャップロックと競合します。

```
1:UPDATE:INSERT INTO emp (empno, ename) VALUES (7784, 'steve')
```

4. TX2のINSERT文が7788(scott)の手前のギャップに対して挿入インテンションギャップロックの取得を試み、TX1のギャップロックと競合してデッドロックが発生します。

```
2:UPDATE:INSERT INTO emp (empno, ename) VALUES (7786, 'bill')
```

宿題

- 対処方法を考えてみてください。