PostgreSQLを用いた分散データベースの開発

慶応義塾大学大学院
政策メディア研究科
永安 悟史

Agenda

- PostgreSQLについて
- 二相コミットの設計と実装
- 二相コミットを用いたレプリケーションの設計と実装、評価
- 分散トランザクションの設計と実装

PostgreSQLとは

- オープンソースRDBMS
  - http://www.jp.postgresql.org/
  - コア開発者6人、メイン開発者14人（日本人2）
  - RDBMSに必要な機能はおおむね実装済み
  - 商用製品で言うと Oracle, MSSQL Server など
  - 特に日本では圧倒的な人気
  - あとは細かな改良

他RDBMSとの機能比較

	PostgreSQL 7.1	商用DB	MySQL
トランザクション	○	○	△
MVCC	○	△	×
行ロック	○	○	×
外部キー	○	○	×
サブクエリー	○	○	×
外部結合	○	○	○
ユーザー定義データ型	○	△	×
ストアドプロシージャ	○	○	×
トリガー	○	○	×
ルール	○	×	×
分散データベース	×	○	×
レプリケーション	×	△	△
トランザクションログ	○	○	×
マルチバイト対応	○	△	△
オープンソース・フリーソフト	○	×	○

PostgreSQLの問題点

- レプリケーション（複製）機能が無い
  - ほとんど問題は起きないが、起きたときが問題。「転んでも走りつづける」システムが必要。
  - 24時間Non-stopシステムの構築が困難
- 分散データベース機能が無い
  - 大規模&高負荷データベースを構築できない
- PCクラスタ、ブレードサーバ普及への対応
  - スケールアウト

「分散データベース」とは

- 広義には
  - 同期レプリケーションを含む、分散したノード上で協調動作するデータベースシステム
- 狭義には
  - 複数のノード上に、ひとつのデータベース、あるいはテーブルを分割配置し、ひとつのトランザクションの中で必要なデータを利用できる。

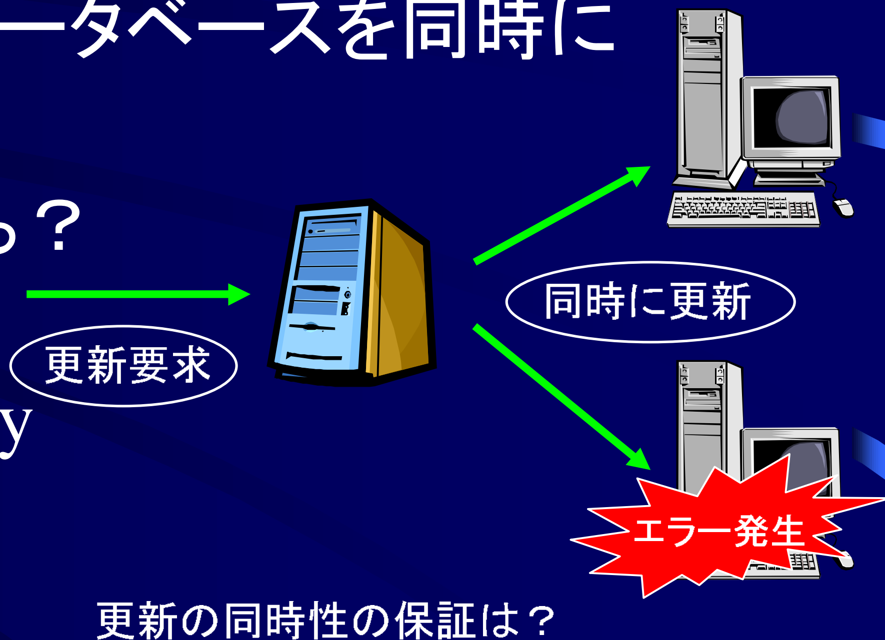
分散データベースの実現には

- 分散したノード間で協調して動作することが必要
- トランザクションのACID属性
  - Atomicity（原子性）、Consistency（一貫性）、Isolation（分離性）、Durability（永続性）
- 二相コミット（2-phase commit）を実装
  - 分散したノード間でのトランザクション処理

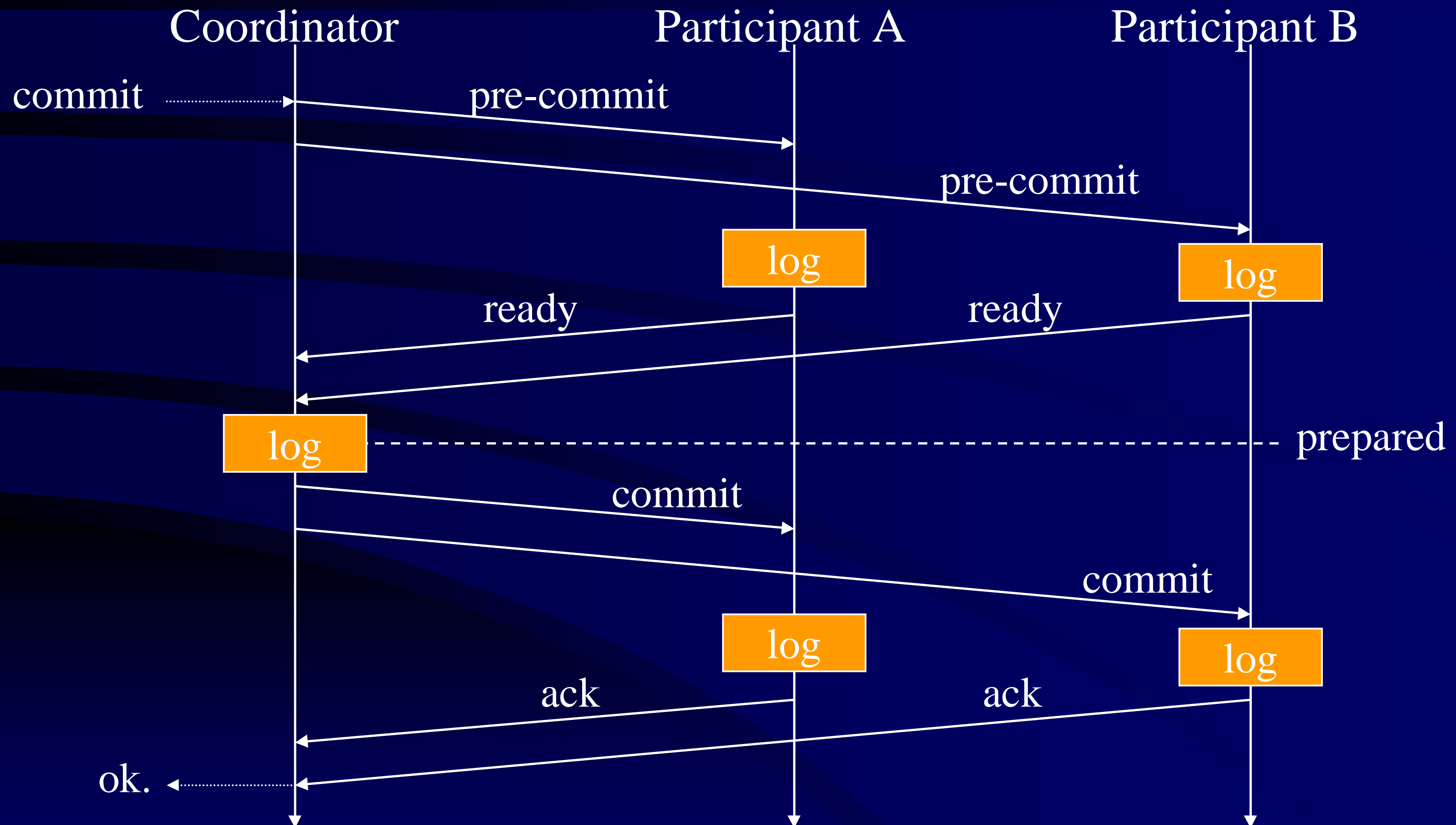
二相コミットの設計と実装

二相コミットプロトコルとは

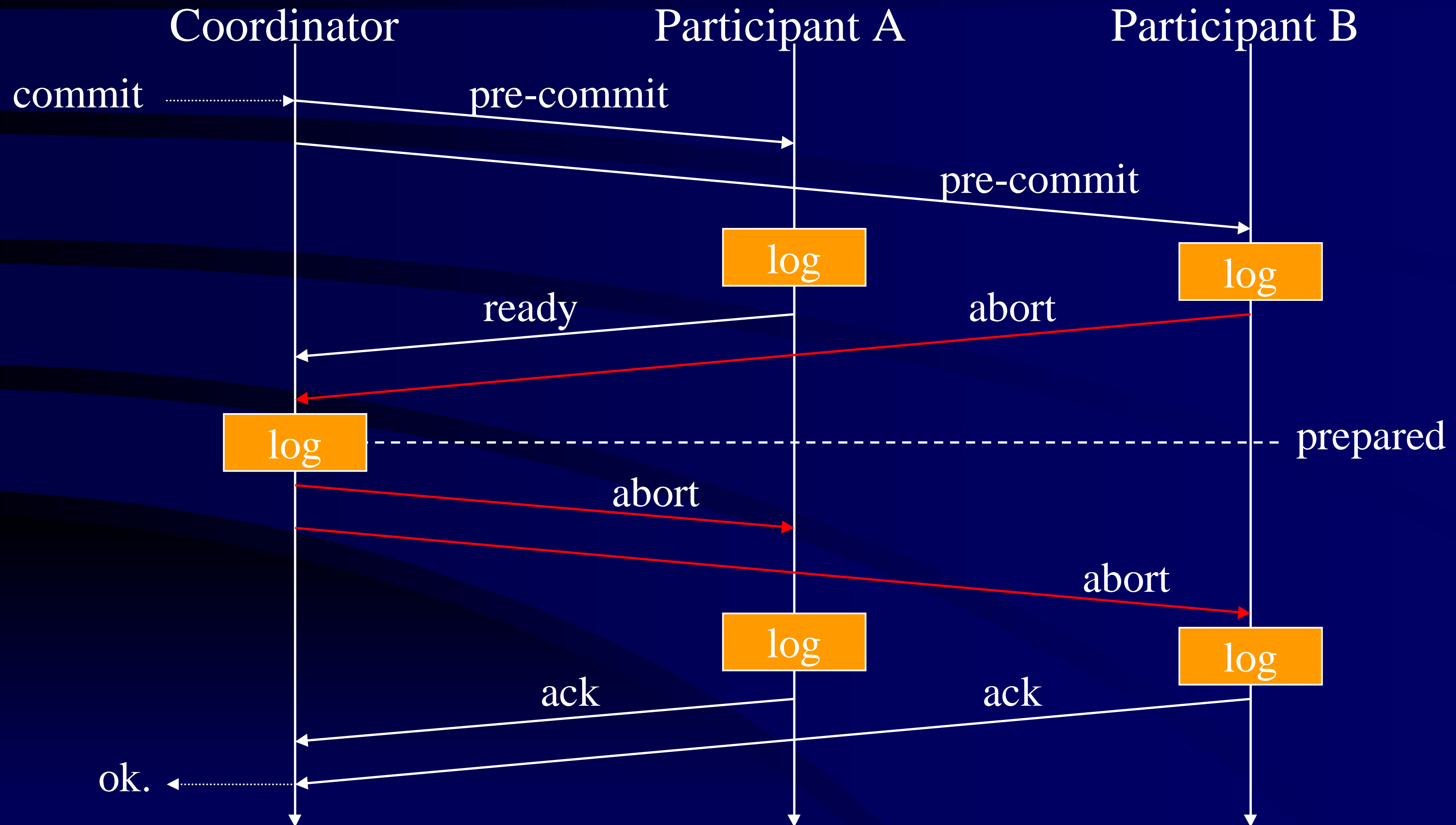
- 二相コミット（2-phase commit）プロトコル
  - 分散したノード上のデータベースを同時に更新する
  - 一部だけが失敗したら？
  - トランザクションのAtomicity、Consistensy



二相コミットプロトコル



二相コミットプロトコル



二相コミットのメリットとデメリット

- メリット
  - 理解しやすい
  - 実装しやすい
  - リカバリしやすい
  - オープン・スタンダード（X/OpenのXA）
- デメリット
  - 一番遅いノードにパフォーマンスがひきずられる

二相コミットの設計と実装

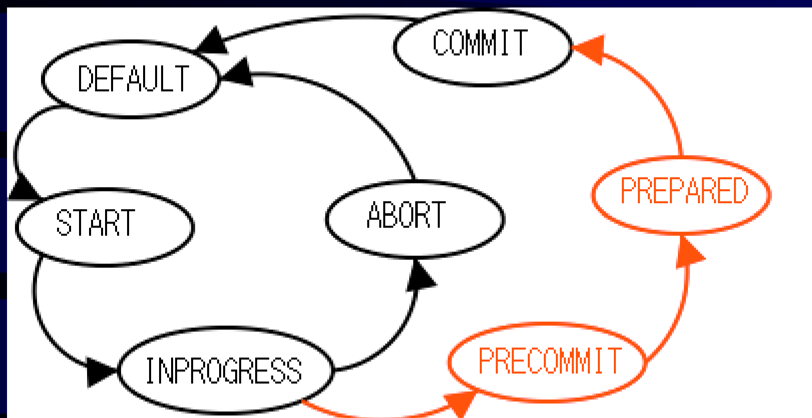
- 二相コミットの必要な機能
  - Pre-commit状態の追加・保持
  - Pre-commitのlogging
  - コミット処理の分離（第一相と第二相）
  - プロトコルの拡張

二相コミットの設計と実装

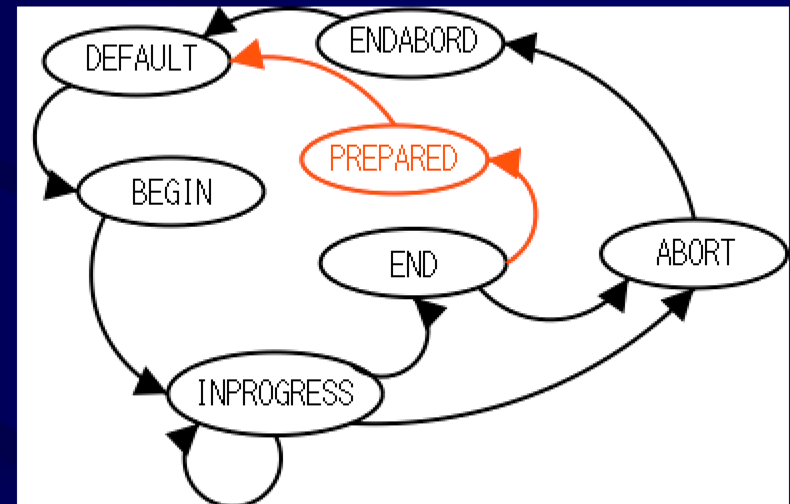
- トランザクション状態変数の追加
  - TransactionState
    - TRANS_DEFAULT, _START, _INPROGRESS, _COMMIT, _ABORT, **_PRECOMMIT**, **_PREPARED**
  - TransactionBlockState
    - TBLOCK_DEFAULT, _BEGIN, _INPROGRESS, _END, _ENDABORT, _ABORT, **_PREPARED**
  - TransactionLog(XLOG) / CommitLog (CLOG)
    - **XLOG_XACT_REPRECOMMIT 0x10**
    - **TRANSACTION_STATUS_PRECOMMITTED 0x02**

トランザクションの状態遷移

TransactionState



TransactionBlockState



二相コミットの設計と実装

- CommitTransaction()を二相に分離
  - PreCommitTransaction()
  - CommitTransaction()
- RecordTransactionPreCommit()の実装

二相コミットの設計と実装

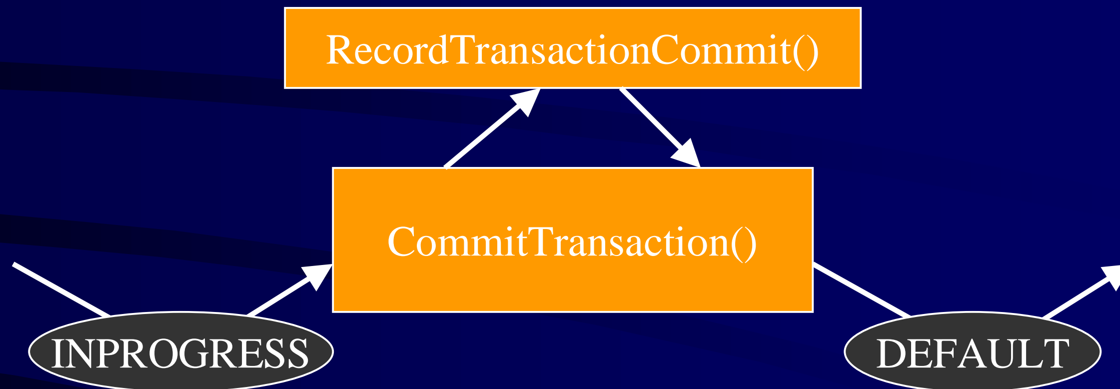
```
PreCommitTransaction(void)
{
    TransactionState s = CurrentTransactionState;

    if (s->state != TRANS_INPROGRESS)
        elog(WARNING, "PreCommitTransaction and not in in-progress state");

    DeferredTriggerEndXact();
    HOLD_INTERRUPTS();

    s->state = TRANS_PRECOMMIT;

    lo_commit(true);
    AtCommit_Notify();
    AtEOXact_portals();

    RecordTransactionPreCommit();

    s->state = TRANS_COMMIT_READY;

    RESUME_INTERRUPTS();

    elog(LOG, "PreCommitTransaction done. xid=%x", s->transactionIdData);
}
```

二相コミットの設計と実装

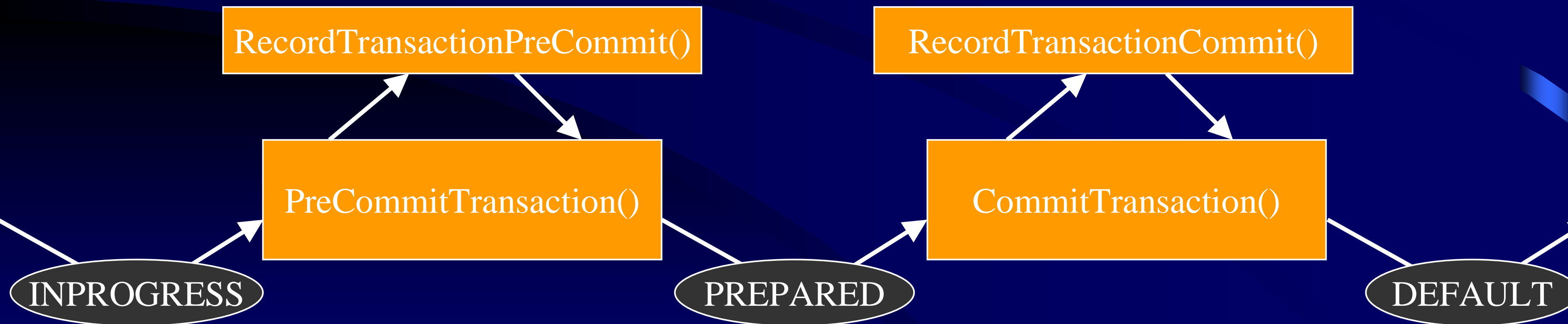
```
CommitTransaction(void)
{
    TransactionState s = CurrentTransactionState;

    if (s->state != TRANS_COMMIT_READY)
        elog(WARNING, "not in commit-ready state");

    HOLD_INTERRUPTS();

    s->state = TRANS_COMMIT;

    RecordTransactionCommit();

    if (MyProc != (PGPROC *) NULL)
    {
        LWLockAcquire(SInvalLock, LW_EXCLUSIVE);
        MyProc->xid = InvalidTransactionId;
        MyProc->xmin = InvalidTransactionId;
        LWLockRelease(SInvalLock);
    }

    smgrDoPendingDeletes(true);

    AtEOXact_GUC(true);
    AtEOXact_SPI();
    AtEOXact_gist();
    AtEOXact_hash();
    AtEOXact_nbtree();
    AtEOXact_rtree();
    AtEOXact_Namespace(true);
    AtCommit_Cache();
    AtCommit_Locks();
    AtEOXact_CatCache(true);
    AtCommit_Memory();
    AtEOXact_Buffers(true);
    smgrabort();
    AtEOXact_Files();

    pgstat_count_xact_commit();

    s->state = TRANS_DEFAULT;

    RESUME_INTERRUPTS();

    elog(LOG, "CommitTransaction done. xid=%x", s->transactionIdData);
}
```

二相コミットまとめ

- 今まで



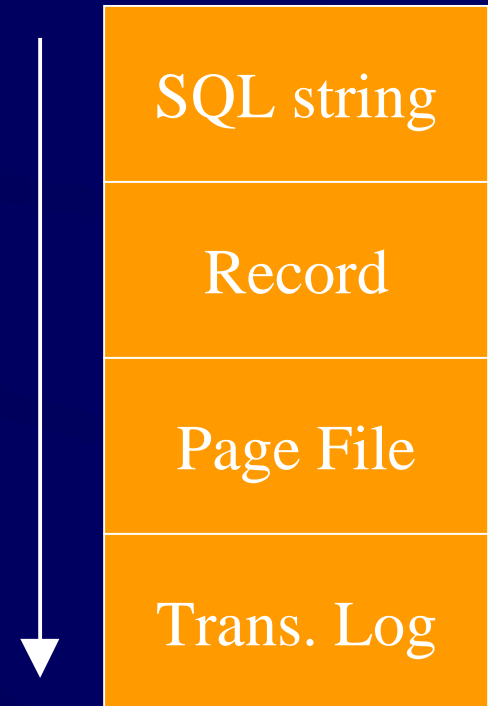
- 現在



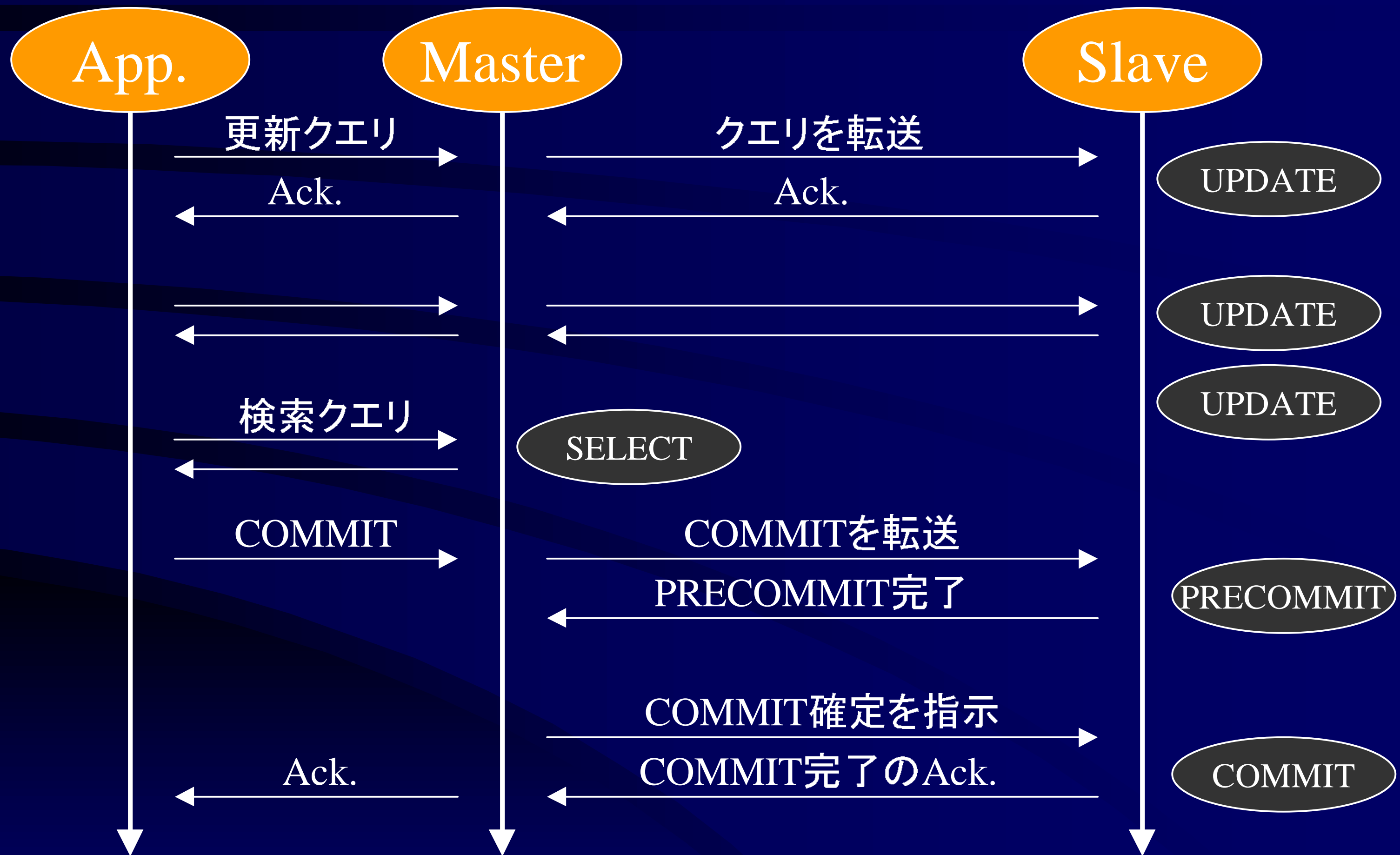
レプリケーションの設計と実装

レプリケーションの種類

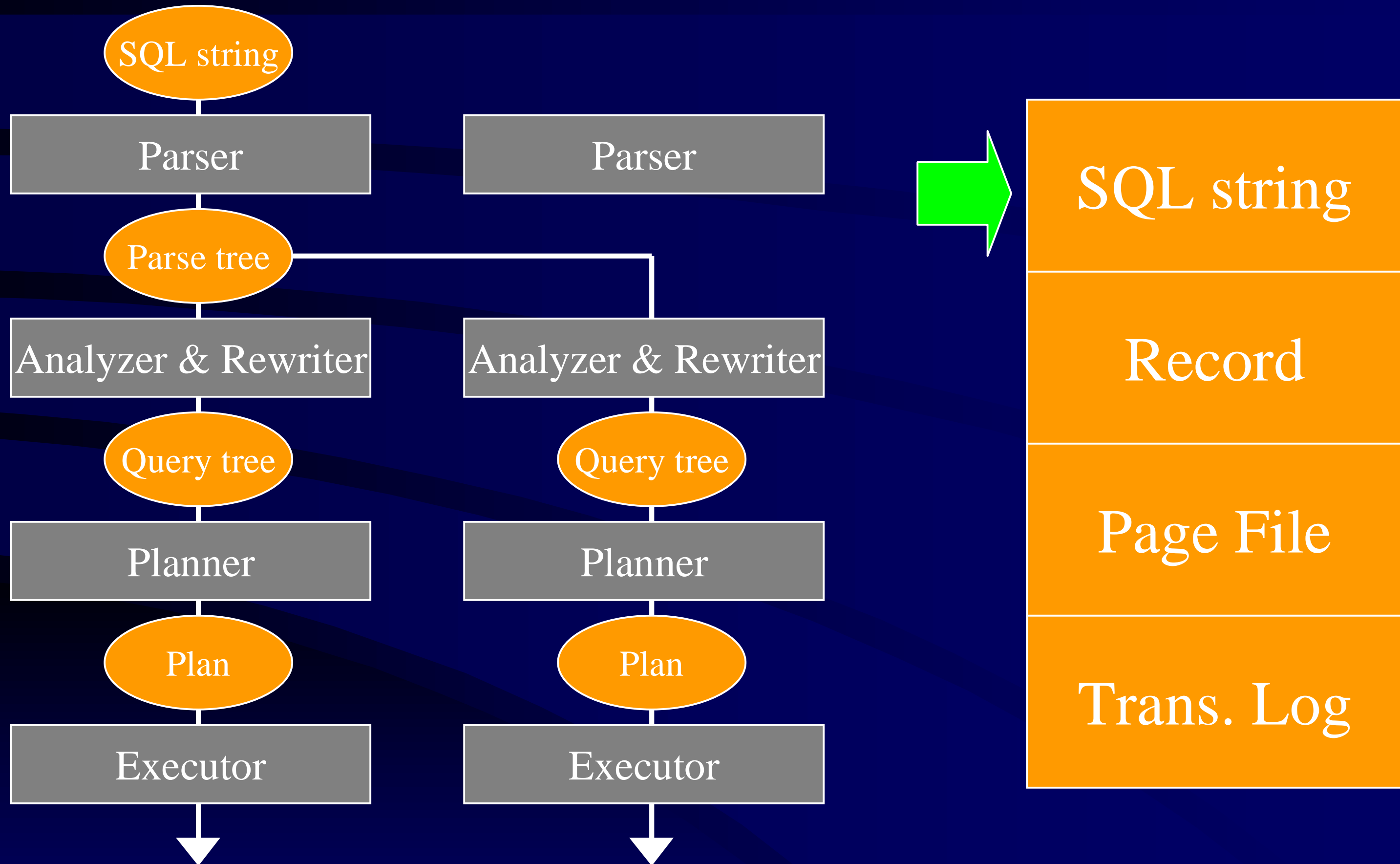
- クエリ(SQL)
  - SQLの文字列を各ノードに転送
- レコード、ディスクI/O
  - I/Oに反映する直前で複製
- トランザクションログ
  - ログを用いて内容を再構成
- 設計のポリシー次第



レプリケーションの設計



レプリケーションの設計



クエリレベル・レプリケーション

- メリット
  - 仕組みがシンプルで実装が簡単
- デメリット
  - ノードごとに異なる値になるSQL関数の問題（CURRENT_TIMESTAMP, rand()など）
  - ノード間のsequenceの整合性

プロトコルの拡張

レプリケーションの実装

PostgresMain()
pg_exec_query_string()

[1st phase]

finish_xact_command()
CommitTransactionCommand()
{ DEFAULT | INPROGRESS }
PreCommitTransaction()
PrepareReplica()
RecordReplicaPrepared()

‘Q’commit;
‘L’COMMIT

PostgresMain()

pg_exec_query_string()
finish_xact_command()
CommitTransactionCommand()
PreCommitTransaction()

[2nd phase]

finish_xact_command()
CommitTransactionCommand()
{ PREPARED }
PreCommitTransaction()
CommitReplica()
RecordReplicaCommitted()

‘L’COMMIT
‘C’COMMIT

CommitTransactionCommand()
CommitTransaction()

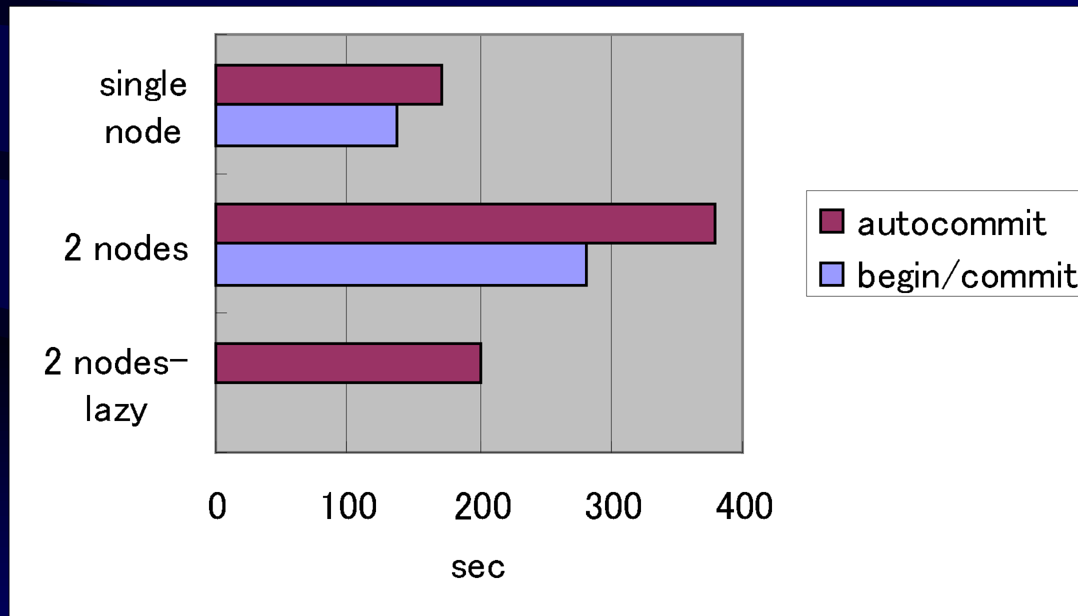
パフォーマンス評価

- 10,000回のUPDATE（更新処理）

```
CREATE TABLE t1 ( uid integer primary key, count integer );
INSERT INTO t1 VALUES ( 1, 0 );
UPDATE t1 SET count = 0 WHERE uid = 1;
UPDATE t1 SET count = 1 WHERE uid = 1;
UPDATE t1 SET count = 2 WHERE uid = 1;
…
UPDATE t1 SET count = 9998 WHERE uid = 1;
UPDATE t1 SET count = 9999 WHERE uid = 1;
```

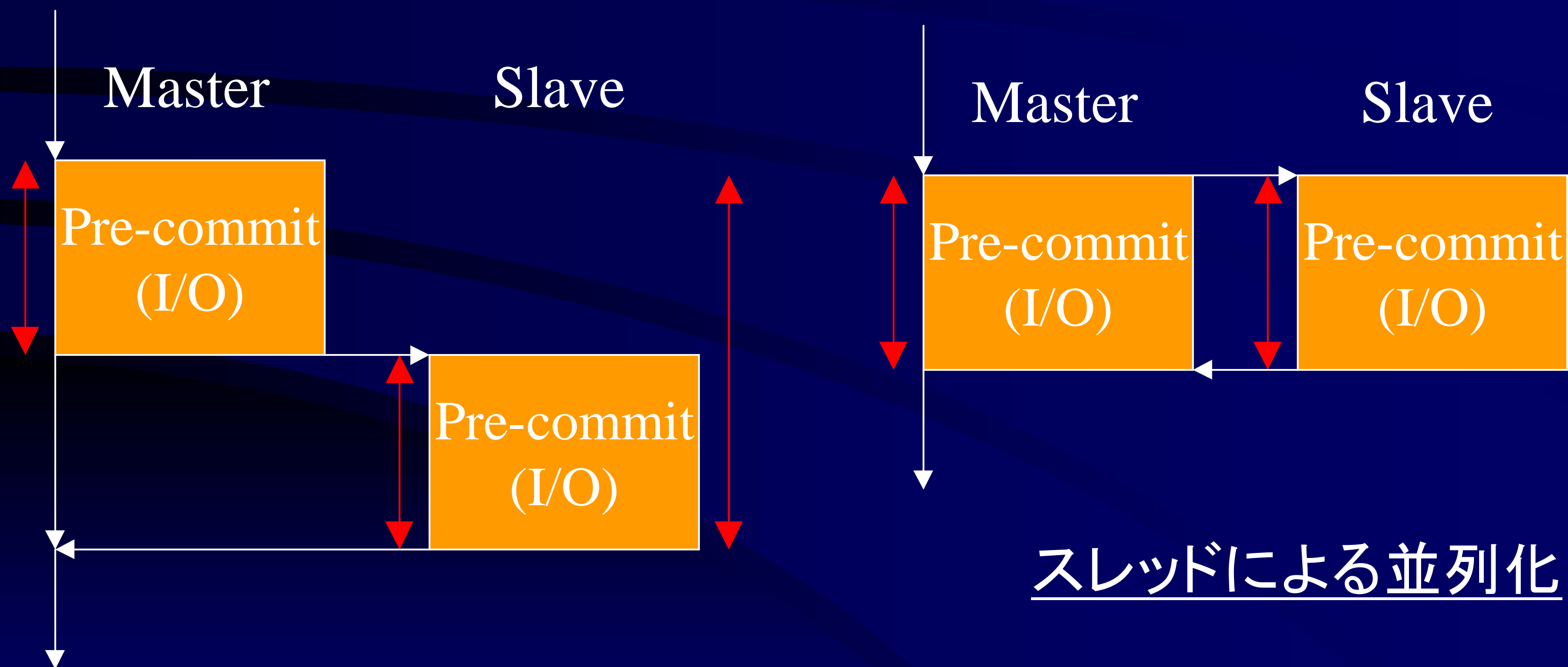
パフォーマンス評価

- UPDATE 10,000回
- COMMITは10,000回&1回



ボトルネックと改善策

- 各ノード上のPre-commitのシーケンス



レプリケーションまとめ

- 二相コミットを用いたレプリケーションを実装した
- パフォーマンスに問題
  - Loggingの実装の改善で対処可能
- 二相コミットのエラーリカバリ（リスタート）を実装する必要

分散トランザクションの設計と実装

分散トランザクション

- 分散トランザクションとは
  - データの分散しているサイト（ノード）をまたぐトランザクション
  - トランザクションのACID属性
  - 遠隔地のDBをひとつのトランザクションで利用したい

分散トランザクション

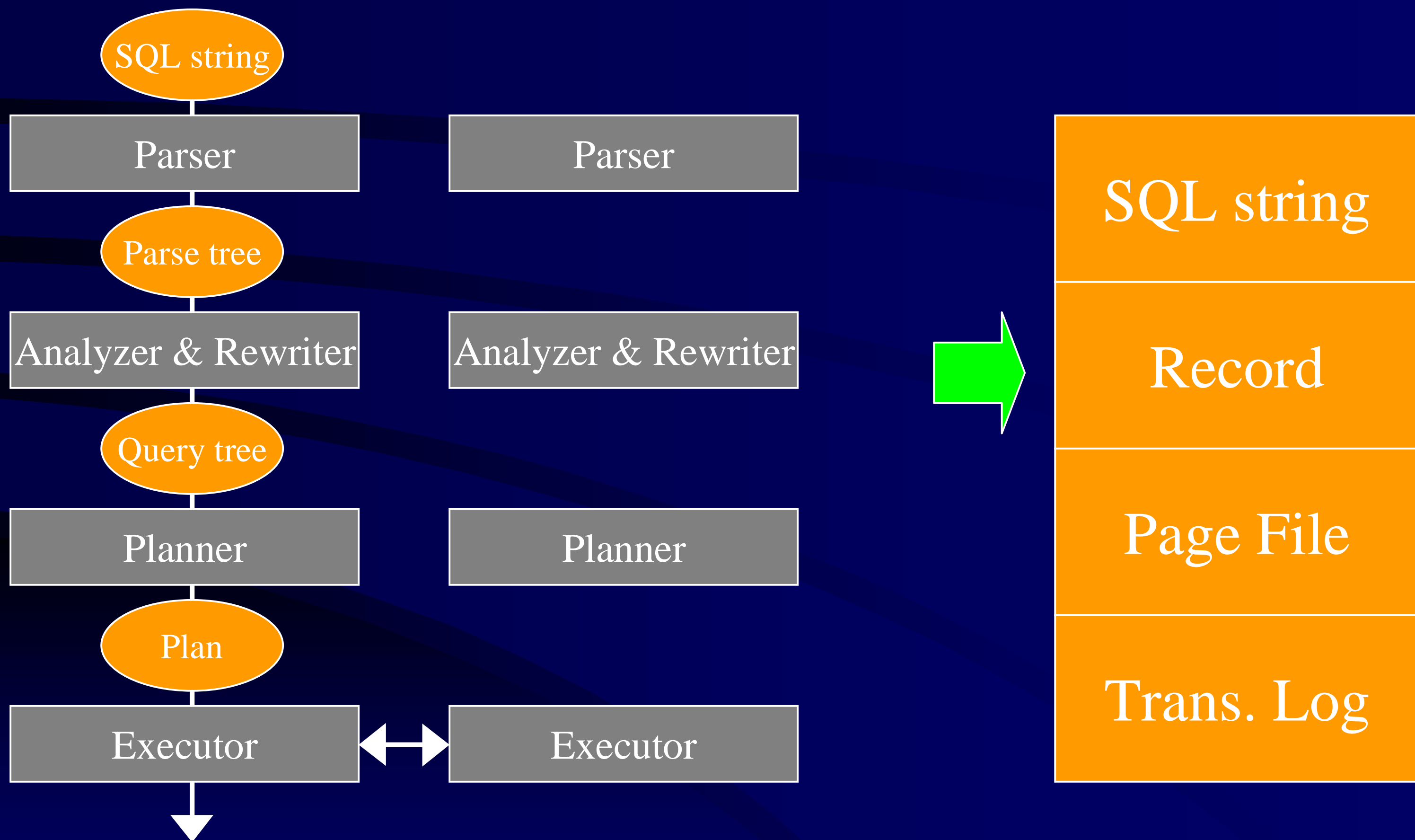
- 例
  - 大阪に在庫がない場合、名古屋に問い合わせ、それでもない場合、東京へ

```
BEGIN;
SELECT itemId FROM zaiko@osaka WHERE numItem<10;
IF ( SELECT numItem FROM zaiko@nagoya WHERE itemId = id ) > 100
    INSERT INTO order@nagoya (itemId, number)
                        VALUES (id, 100);
ELSE
    INSERT INTO order@tokyo (itemId, number)
                        VALUES (id, 100);
COMMIT;
```

分散トランザクションの設計と実装

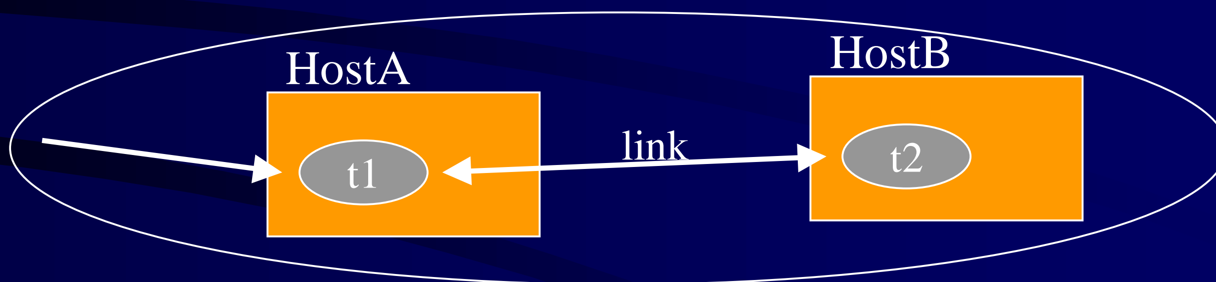
- 分散トランザクションの実現に必要な機能
  - Executorがネットワーク経由でリモートサイトのテーブルやインデックスをレコード単位でダイレクトに使う（RPC機能）
  - トランザクションの単一性（A）、一貫性（C）
    - 二相コミットを利用
  - データの分散配置を考慮したオプティマイザ

分散トランザクションの設計と実装



Parserの拡張

- テーブル名の拡張表現(リンク名)を許容
  - SELECT * FROM t1, t2@link WHERE …

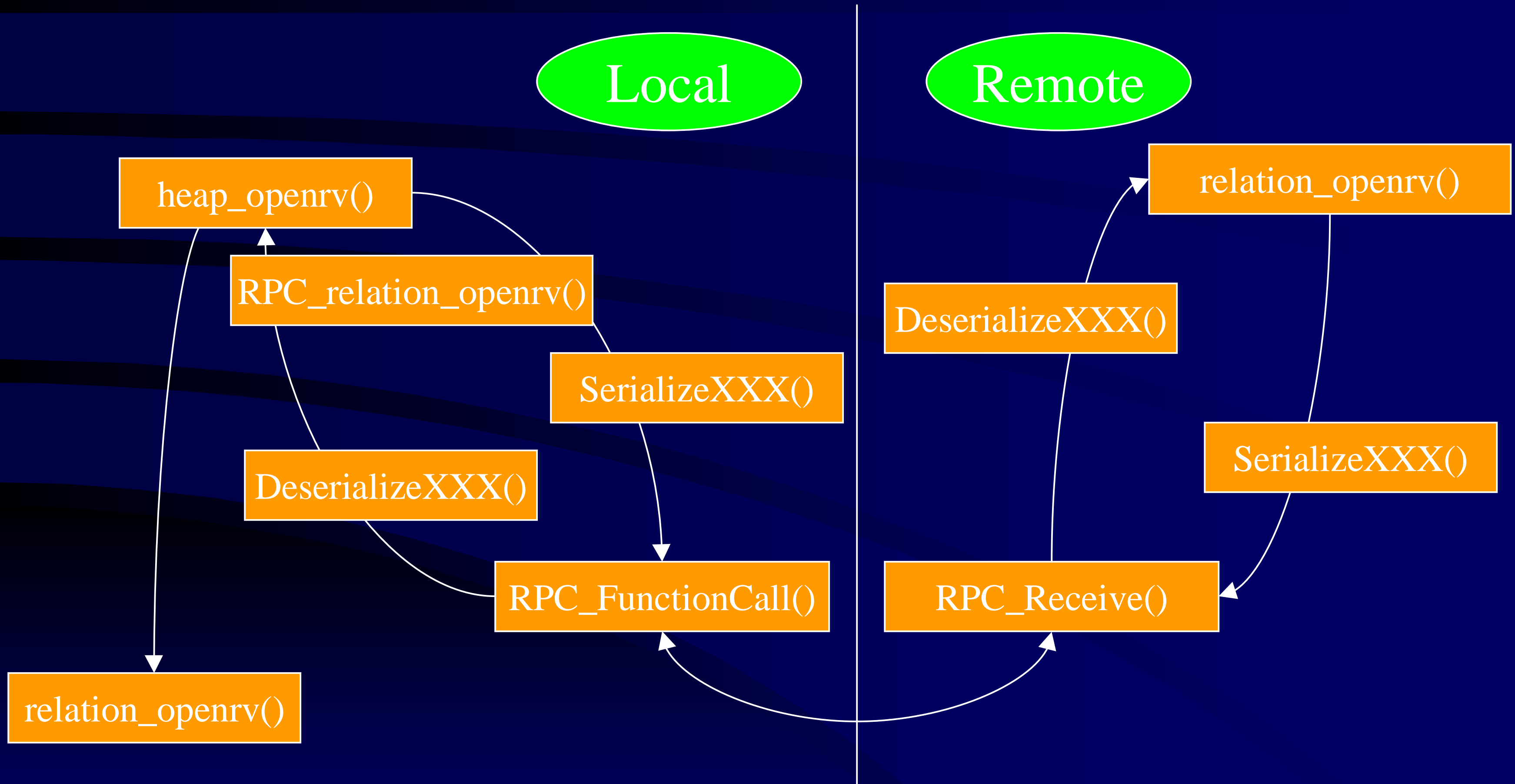


  - RangeVar構造体(テーブル表現のデータ構造)にリンク名を格納 → Analyze&Rewriteへ

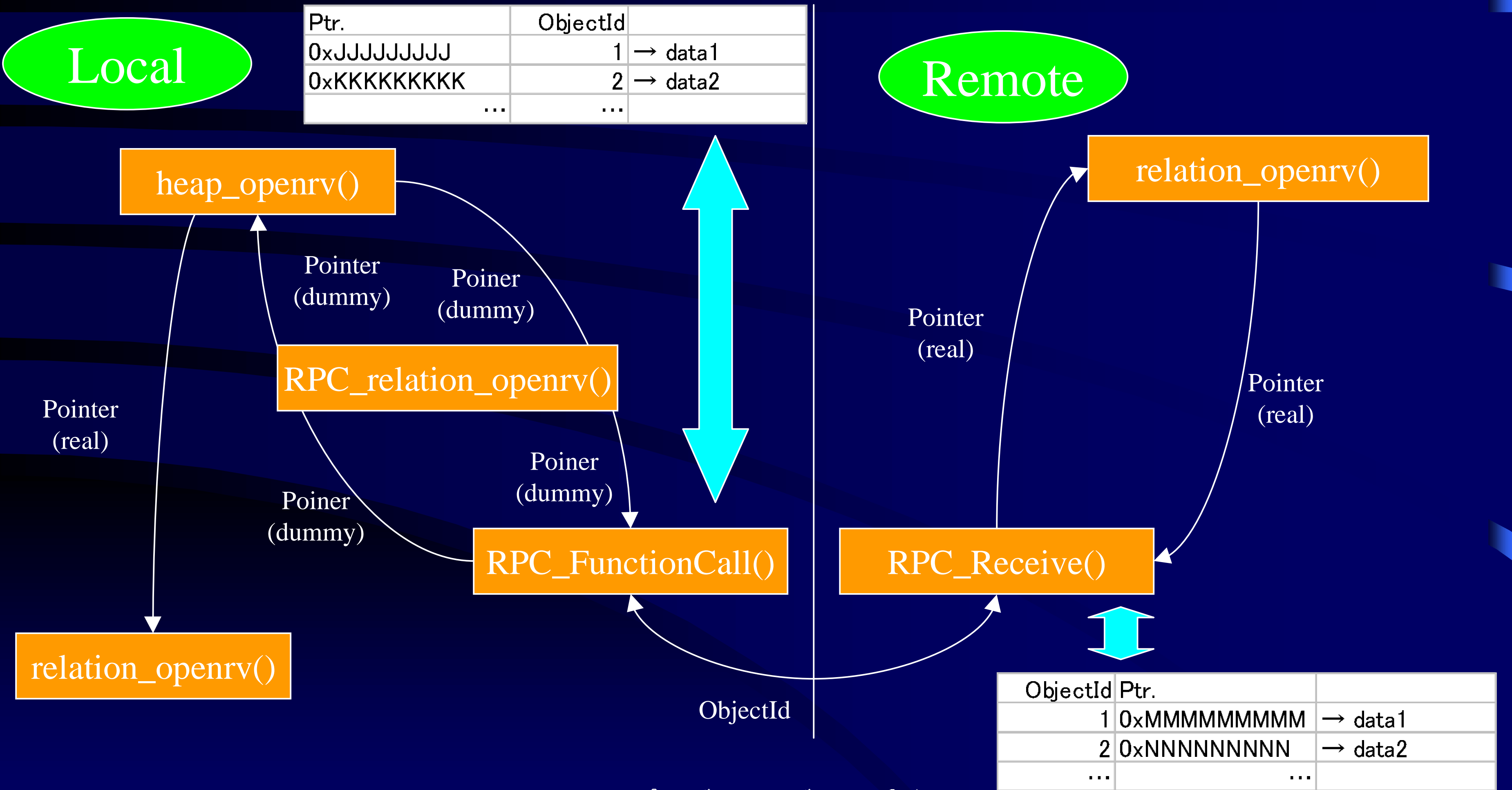
アクセスメソッドRPC

- ヒープへのアクセスメソッドをRPC化
  - relation_openrv() → RPC_relation_openrv()
- オブジェクト(構造体)をシリアライズ
  - 連続したメモリブロックに変換
  - SerializeXXX(), DeserializeXXX()
- シリアライズしたオブジェクトを転送
  - RPC_FunctionCall()
  - Executor間で通信

シリアライズによる通信



オブジェクトIDによる通信



オプティマイザ

- まったく未着手
  - 分散環境における最適化はゼロから．．．
  - 当面はオプティマイズ無しで．．．

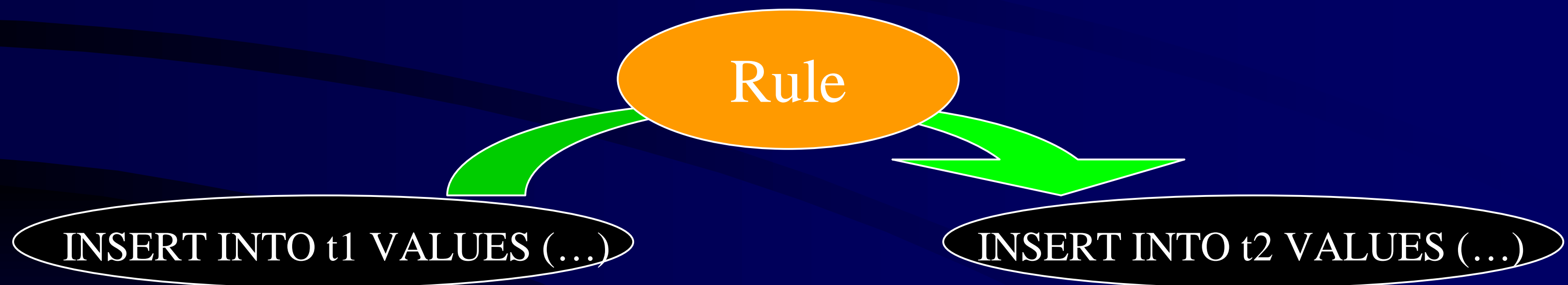
分散トランザクションまとめ

- Parserの最低限の拡張はできた
- RPCアクセスメソッドを実装中
  - 「内部のAPI」と呼べるような層がない…感じ。どれをRPC化すればいいのか？
  - レプリケーションもできる
- シリアライズが難しい
  - ダイナミックなメモリ割り当てが非常に多い
  - リンクリスト構造が非常に多い

今後の（希望的）予定

ルール（Rewrite Rule）

- 指定した規則に基づいてクエリを書き換え
  - CREATE RULE rule_name AS ON event
  - TO object [WHERE rule_qualification]
  - DO [INSTEAD] [action | (actions) | NOTHING];



- 分散トランザクションにおいて、データの配置を制御することが可能

テーブルパーティショニング

– 大規模＆負荷分散

uid	name	address
1		
2		
3		
…		
49998		
49999		
50000		
50001		
50002		
…		

UPDATE …
WHERE uid=47015

SELECT …
WHERE uid=50023

if uid<50000 then DB1
else DB2

DB 1

DB 2

自律的リソースバランシング

- 負荷状況、レスポンスタイム、ストレージ状況に応じた動的なデータ配置の変更
  - データ配置の変更はルールの書き換え（＋α）で対応可能
  - 突発的にアクセスが殺到する場合に備える

謝辞

- IPA未踏ユース「PostgreSQLを用いた分散データベースの開発」

- 日本PostgreSQLユーザー会・分散トランザクション開発分科会

終