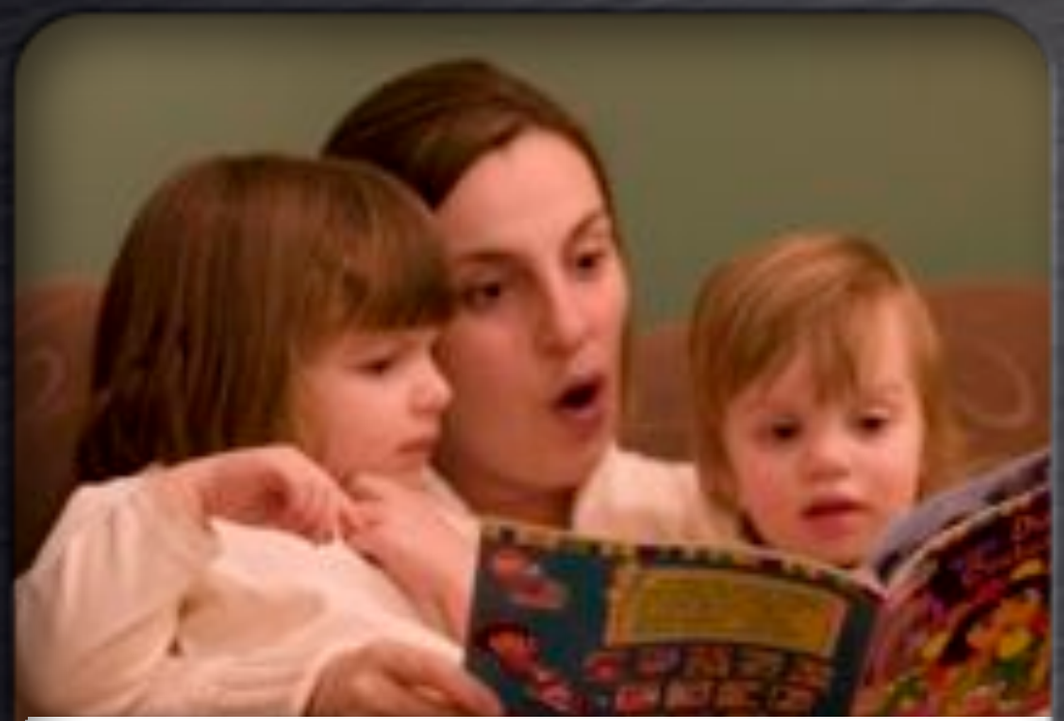
Big Bad PostgreSQL

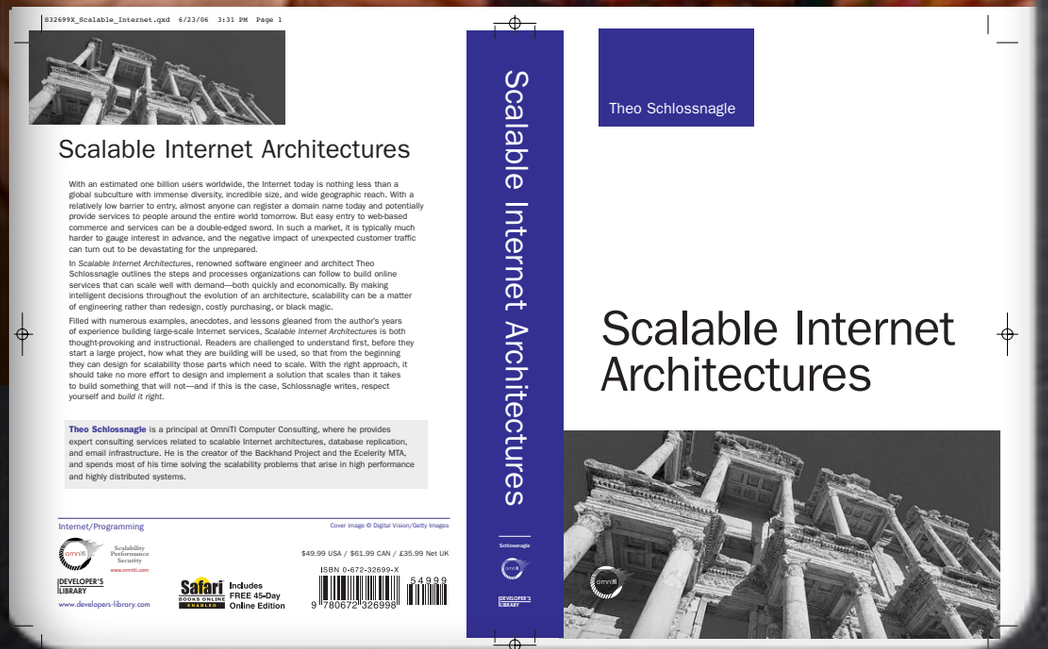
A Case Study

omniti

Moving a
"large,"
"complicated," and
mission-critical
datawarehouse
from ***Oracle***
to ***PostgreSQL***
for ***cost control***.

About the Speaker





- Principal @ OmniTI
- Open Source

  mod_backhand, spreadlogd, OpenSSH+SecurID, Daiquiri, Wackamole, libjlog, Spread, etc.
- Closed Source

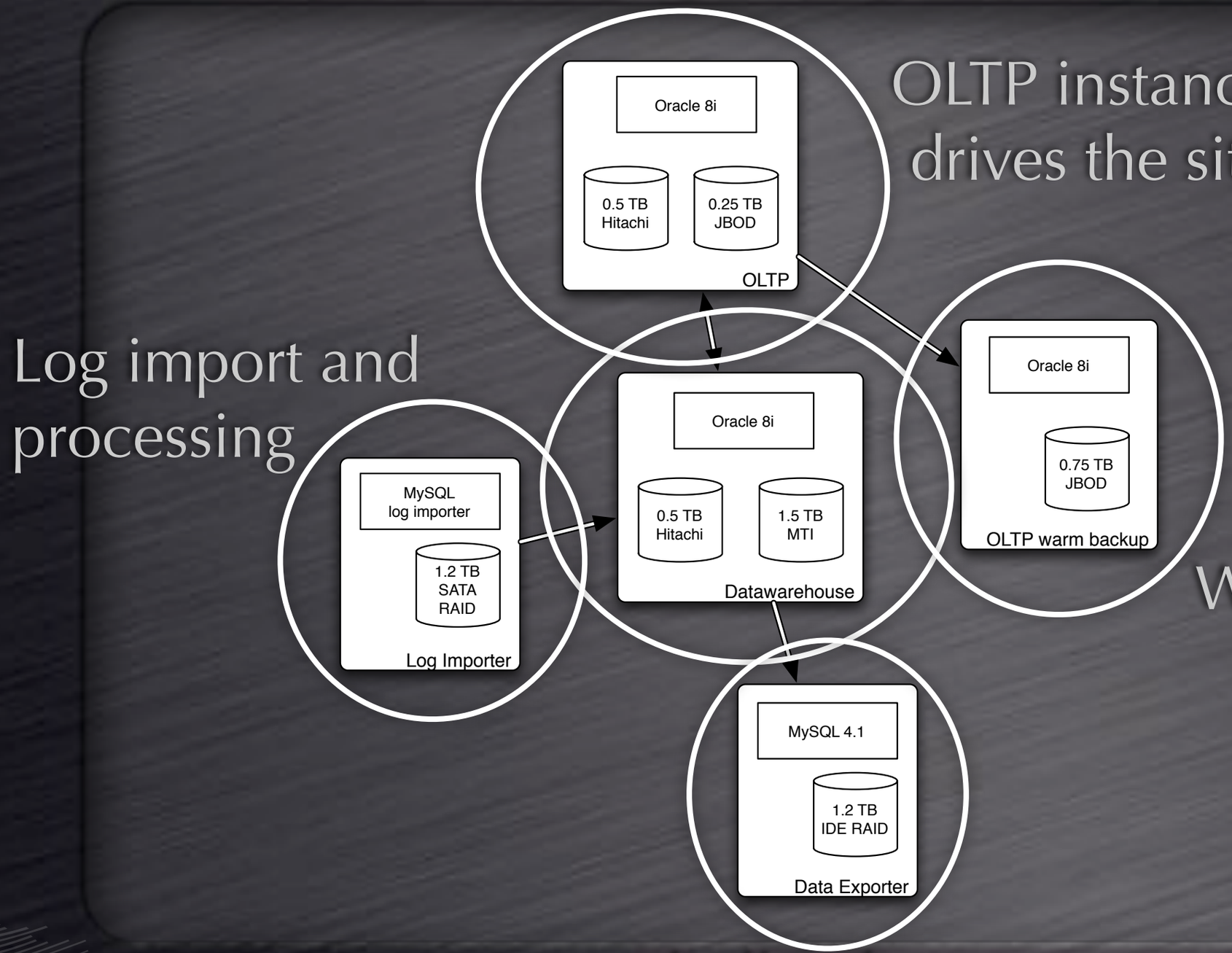
  Ecelerity and EcCluster
- Author

  Scalable Internet Architectures

Glossary

- OLTP
  - Online Transaction Processing
- ODS
  - Operational Datastore
  - (a.k.a. Data Warehouse)

Overall Architecture



Database Situation

- The problems:
  - The database is growing.
  - The OLTP and ODS/warehouse are too slow.
  - A **lot** of application code against the OLTP system.
  - Minimal application code against the ODS system.
- Oracle:
  - Licensed per processor.
  - Really, really, really expensive on a large scale.
- PostgreSQL:
  - No licensing costs.
  - Good support for complex queries.

Database Choices

- Must keep Oracle on OLTP
  - Complex, Oracle-specific web application.
  - Need more processors.
- ODS: Oracle not required.
  - Complex queries from limited sources.
  - Needs more space and power.
- Result:
  - Move ODS Oracle licenses to OLTP
  - Run PostgreSQL on ODS

PostgreSQL gotchas

- For an OLTP system that does thousands of updates per second, vacuuming is a hassle.
- No upgrades?!
- Less community experience with large databases.
- Replication features less evolved.

PostgreSQL ♥ ODS

- Mostly inserts.
- Updates/Deletes controlled, not real-time.
- pl/perl (leverage DBI/DBD for remote database connectivity).
- Monster queries.
- Extensible.

Choosing Linux

- Popular, liked, good community support.
- Chronic problems:
  - kernel panics
  - filesystems remounting read-only
  - filesystems don't support snapshots
  - LVM is clunky on enterprise storage
  - 20 outages in 4 months

Choosing Solaris 10

- Switched to Solaris 10
  - No crashes, better system-level tools.
    - prstat, iostat, vmstat, smf, fault-management.
  - ZFS
    - snapshots (persistent), BLI backups.
  - Excellent support for enterprise storage.
  - DTrace.
  - Free (too).

Oracle features we need

- Partitioning
- Statistics and Aggregations
  - rank over partition, lead, lag, etc.
- Large selects (100GB)
- Autonomous transactions
- Replication from Oracle (to Oracle)

Partitioning

For large data sets:

```
pgods=# select count(1) from ods.ods_tblpick_super;
   count
------------
 1790994512
(1 row)
```

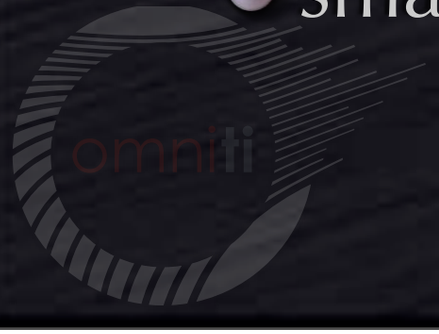
- Next biggest tables: 850m, 650m, 590m
- Allows us to cluster data over specific ranges (by date in our case)
- Simple, cheap archiving and removal of data.
- Can put ranges used less often in different tablespaces (slower, cheaper storage)

Partitioning PostgreSQL style

- PostgreSQL doesn't support partition...
- It supports inheritance... (what's this?)
  - some crazy object-relation paradigm.
- We can use it to implement partitioning:
  - One master table with no rows.
  - Child tables that have our partition constraints.
  - Rules on the master table for insert/update/delete.

Partitioning PostgreSQL realized

- Cheaply add new empty partitions
- Cheaply remove old partitions
- Migrate less-often-accessed partitions to slower storage
- Different indexes strategies per partition
- PostgreSQL >8.1 supports constraint checking on inherited tables.
  - smarter planning
  - smarter executing



RANK OVER PARTITION

- In Oracle:

```
select userid, email from (
          select u.userid, u.email,
          row_number() over
                 (partition by u.email order by userid desc) as position
          from (...)) where position = 1
```

- In PostgreSQL:

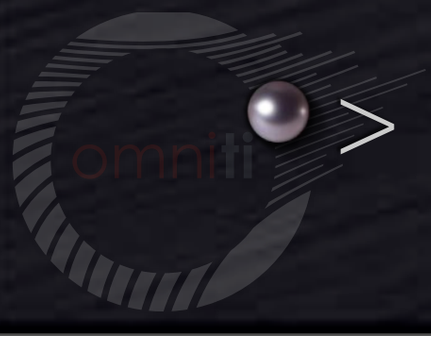
```
FOR v_row IN select u.userid, u.email from (...) order by email, userid desc
LOOP
        IF v_row.email != v_last_email THEN
          RETURN NEXT v_row;
          v_last_email := v_row.email;
          v_rownum := v_rownum + 1;
        END IF;
END LOOP;
```

Large SELECTs

- Application code does:

```
select u.*, b.browser, m.lastmess
  from ods.ods_users u,
       ods.ods_browsers b,
       (  select userid, min(senddate) as senddate
            from ods.ods_maillog
        group by userid ) m,
       ods.ods_maillog l
where u.userid = b.userid
  and u.userid = m.userid
  and u.userid = l.userid
  and l.senddate = m.senddate;
```

- The width of these rows is about 2k
- 50 million row return set
- > 100 GB of data



The Large SELECT Problem

- libpq will buffer the ***entire*** result in memory.
  - This affects language bindings (DBD::Pg).
  - This is an utterly deficient default behavior.
- This can be avoided by using cursors
  - Requires the app to be PostgreSQL specific.
  - You open a cursor.
  - Then FETCH the row count you desire.

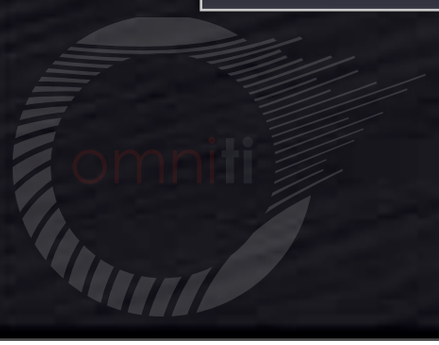
Big SELECTs the Postgres way

The previous "big" query becomes:

```
DECLARE CURSOR bigdump FOR
select u.*, b.browser, m.lastmess
  from ods.ods_users u,
       ods.ods_browsers b,
       (  select userid, min(senddate) as senddate
            from ods.ods_maillog
        group by userid ) m,
       ods.ods_maillog l
 where u.userid = b.userid
   and u.userid = m.userid
   and u.userid = l.userid
   and l.senddate = m.senddate;
```

Then, in a loop:

```
FETCH FORWARD 10000 FROM bigdump;
```



Autonomous Transactions

- In Oracle we have over 2000 custom stored procedures.
- During these procedures, we like to:
  - COMMIT incrementally
    Useful for long transactions (update/delete) that need not be atomic -- incremental COMMITs.
  - start a new **top-level** txn that can COMMIT
    Useful for logging progress in a stored procedure so that you know how far you progessed and how long each step took even if it rolls back.

PostgreSQL shortcoming

- PostgreSQL simply does not support Autonomous transactions and to quote core developers “that would be hard.”
- When in doubt, use brute force.
- Use pl/perl to use DBD::Pg to connect to ourselves (a new backend) and execute a new top-level transaction.

Replication

- Cross vendor database replication isn't too difficult.
- Helps a lot when you can do it **inside** the database.
- Using dbi-link (based on pl/perl and DBI) we can.
  - We can connect to any remote database.
  - INSERT into local tables directly from remote SELECT statements.
    [snapshots]
  - LOOP over remote SELECT statements and process them row-by-row.
    [replaying remote DML logs]

Snapshot mapping

```
pgods=# \d avail.snapshot_tbltranslation
      Table "avail.snapshot_tbltranslation"
   Column    |          Type          | Modifiers
-------------+------------------------+-----------
 src_db      | integer                |
 src_tblname | character varying(255) |
 dst_tblname | character varying(255) |
 col_name    | character varying(255) |
 col_type    | character varying(30)  |
```

Destination tables

```
CREATE OR REPLACE FUNCTION snapshot_create_table_ddl(varchar, varchar) RETURNS text AS $$
DECLARE
  v_dst_tblname ALIAS FOR $1;
  v_suffix ALIAS FOR $2;
  v_create_def TEXT;
  v_index INTEGER;
  v_tbltranslation RECORD;
BEGIN
  v_create_def := 'CREATE TABLE ' || v_dst_tblname || '_' || v_suffix || ' (';
  v_index = 0;
  FOR v_tbltranslation IN SELECT col_name, col_type
                                  FROM snapshot_tbltranslation WHERE
                                  dst_tblname = v_dst_tblname LOOP
    IF v_index > 0 THEN
      v_create_def := v_create_def || ', ';
    END IF;
    v_create_def := v_create_def || ' ' ||
                    '"' || v_tbltranslation.col_name || '" ' ||
                    v_tbltranslation.col_type ;
    v_index := v_index + 1;
  END LOOP;
  v_create_def := v_create_def || ' ) ';
  return v_create_def;
END
$$ LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION snapshot_create_table(varchar, varchar) RETURNS void AS $$
DECLARE
  v_sql text;
BEGIN
  SELECT INTO v_sql snapshot_create_table_ddl($1,$2);
  EXECUTE v_sql;
END;
$$ LANGUAGE 'plpgsql';
```

Performing a snapshot (1)

```
CREATE OR REPLACE FUNCTION perform_snapshot(text) RETURNS varchar AS $$
DECLARE
  v_src_tblname ALIAS FOR $1;
  v_dst_tblname TEXT;
  v_dbi_dsid INTEGER;
  v_index INTEGER;
  v_insert_sql TEXT;
  v_select_sql TEXT;
  v_remote_sql TEXT;
  v_cast_sql TEXT;
  v_qry TEXT;
  v_sql TEXT;
  v_table_exists INTEGER;
  v_job_id INTEGER;
  v_step_id INTEGER;
  v_rowcount INTEGER;
  v_current_snap_tbl VARCHAR;
  v_snap_suffix VARCHAR;
  v_ttrans snapshot_tbltranslation%ROWTYPE;
  v_pds record;
BEGIN
  SELECT INTO v_dst_tblname DISTINCT(dst_tblname)
         FROM snapshot_tbltranslation WHERE src_tblname = v_src_tblname;
  IF v_dst_tblname IS NULL THEN
    RAISE EXCEPTION 'No translation for table %', v_src_tblname;
  END IF;
  SELECT into v_job_id  autonomous_job_log_add_job('' || v_src_tblname);
  v_dbi_dsid := 1;
  v_current_snap_tbl := determine_view_src(v_dst_tblname);

  IF v_current_snap_tbl = 'snap1' THEN
    v_snap_suffix = 'snap2';
  ELSE
    v_snap_suffix = 'snap1';
  END IF;
```

Performing a snapshot (2)

```
select string_to_array(v_dst_tblname, '.') as oparts INTO v_pds;
  select INTO v_table_exists count(1) from pg_tables
    WHERE  schemaname = v_pds.oparts[1] AND
           tablename = v_pds.oparts[2] || '_' || v_snap_suffix;
  IF v_table_exists = 0 THEN
    PERFORM snapshot_create_table(v_dst_tblname, v_snap_suffix);
  ELSE
    SELECT INTO v_step_id
      autonomous_job_log_add_step(
           v_job_id,
           'autonomous truncate and vacuum ' || v_dst_tblname || '_' || v_snap_suffix
    );
   EXECUTE 'select remote_do(3, ''TRUNCATE TABLE ' || v_dst_tblname || '_' || v_snap_suffix || ''')';
   EXECUTE 'select remote_do(3, ''VACUUM FULL ' || v_dst_tblname || '_' || v_snap_suffix || ''')';
   PERFORM autonomous_job_log_upd_step('OK', 'done', v_job_id, v_step_id);
END IF;
```

Performing a snapshot (3)

```
SELECT INTO v_step_id autonomous_job_log_add_step(v_job_id,
                        'snapping into ' || v_dst_tblname || '_' || v_snap_suffix );
v_qry := 'select * from snapshot_tbltranslation where src_tblname = ' ||
         quote_literal(v_src_tblname);
v_insert_sql := 'INSERT INTO ' || v_dst_tblname || '_' || v_snap_suffix || ' (';
v_select_sql := ' SELECT ';
v_remote_sql := 'remote_select(' || v_dbi_dsid || ','''select ';
v_cast_sql := ' t(';
v_index := 0;

FOR v_ttrans IN EXECUTE v_qry LOOP
  IF v_index > 0 THEN
    v_insert_sql := v_insert_sql || ',';
    v_select_sql := v_select_sql || ',';
    v_remote_sql := v_remote_sql || ',';
    v_cast_sql := v_cast_sql || ',';
  END IF;
  v_insert_sql := v_insert_sql || '"' || v_ttrans.col_name || '"';
  v_select_sql := v_select_sql || 't."'|| v_ttrans.col_name || '"';
  v_remote_sql := v_remote_sql || upper(v_ttrans.col_name);
  v_cast_sql := v_cast_sql ||  '"' || v_ttrans.col_name || '" ' ||  v_ttrans.col_type;
  v_index := v_index + 1;
END LOOP;
v_insert_sql := v_insert_sql || ')';
v_select_sql := v_select_sql || ' from ';
v_remote_sql := v_remote_sql || ' from ' || v_src_tblname || ''')';
v_cast_sql := v_cast_sql || ')';

v_sql := v_insert_sql || v_select_sql || v_remote_sql || v_cast_sql;
```

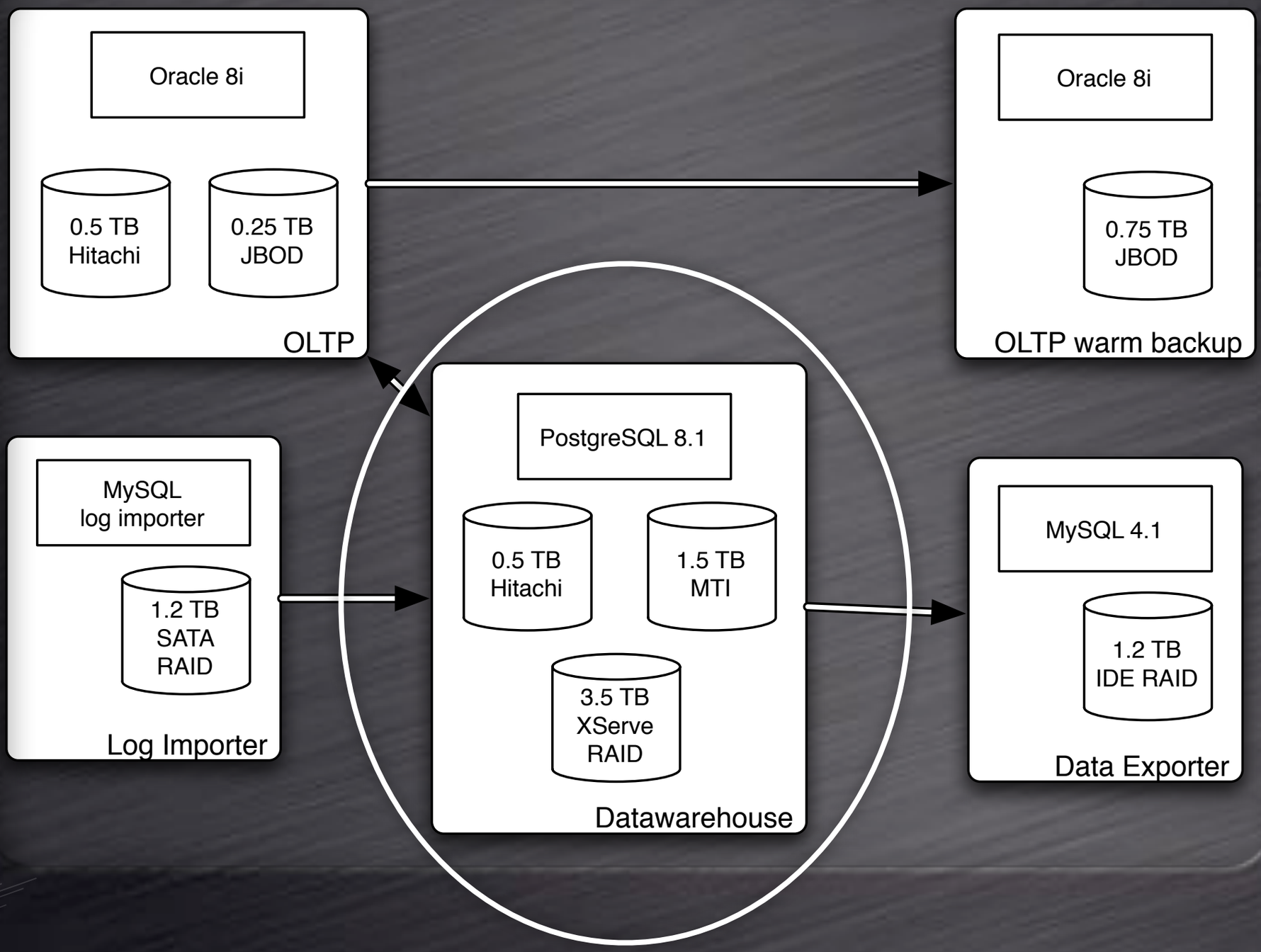
Performing a snapshot (4)

```
  EXECUTE v_sql;
  GET DIAGNOSTICS v_rowcount = ROW_COUNT;
  PERFORM autonomous_job_log_upd_step('OK', 'good (' || v_rowcount ::varchar || ') rows',
                                      v_job_id, v_step_id);
  IF v_rowcount IS NOT NULL THEN
    EXECUTE 'ANALYZE ' || v_dst_tblname || '_' || v_snap_suffix;
    SELECT INTO v_step_id autonomous_job_log_add_step(v_job_id, 'swapping view');
    EXECUTE 'CREATE OR REPLACE VIEW ' || v_dst_tblname || ' AS ' ||
          'SELECT * FROM ' || v_dst_tblname || '_' || v_snap_suffix;
    PERFORM autonomous_job_log_upd_step('OK', 'using  ' || v_dst_tblname || '_' || v_snap_suffix,
                                        v_job_id, v_step_id);
    PERFORM autonomous_job_log_complete_log(v_job_id);
  ELSE
    PERFORM autonomous_job_log_failed_log(v_job_id);
  END IF;
  RETURN v_dst_tblname || '_' || v_snap_suffix;
  EXCEPTION
    WHEN RAISE_EXCEPTION THEN
      RAISE EXCEPTION '%', SQLERRM;
    WHEN OTHERS THEN
      RAISE NOTICE '%', SQLERRM;
      PERFORM autonomous_job_log_upd_step('BAD',
                              'snapshot failed (' || coalesce(SQLERRM, 'unknown error') || ')',
                              v_job_id, v_step_id);
      PERFORM autonomous_job_log_failed_log(v_job_id);
END
$$ LANGUAGE 'plpgsql';
```

Replication (really)

- Through a combination of snapshotting and DML replay we:
  - replicate over into over 2000 tables in PostgreSQL from Oracle
    - snapshot replication of 200
    - DML replay logs for 1800
- PostgreSQL to Oracle is a bit harder
  - out-of-band export and imports

New Architecture



Results

- Move ODS Oracle licenses to OLTP
- Run PostgreSQL on ODS
- Save $500k in license costs.
- Spend $100k in labor costs.
- Learn a lot.