Robust, high-throughput transfer system techniques

Tim Barrass, *University of Bristol, UK*; Daniele Bonacorsi, *INFN-CNAF, Italy*; Jose Hernandez, *CIEMAT, Spain*;
Jens Rehn, *CERN, Switzerland*; Lassi Tuura, *Northeastern University, USA*; Yujun Wu, *Fermilab, USA*

What is PhEDEx?

Building reliable high-performance distributed systems is hard; fortunately much prior art exists. The PhEDEx project [1,2] has sought to apply the best practices known to us. We share here the techniques we have found useful.

PhEDEx is the data placement and transfer system for the CMS experiment [3] at the Large Hadron Collider at CERN [4]. It manages continuous high-load data transfers from CERN to several dozen computing centres around the world; transfers among those centres; and also transfers for individual physicists for their private analyses.

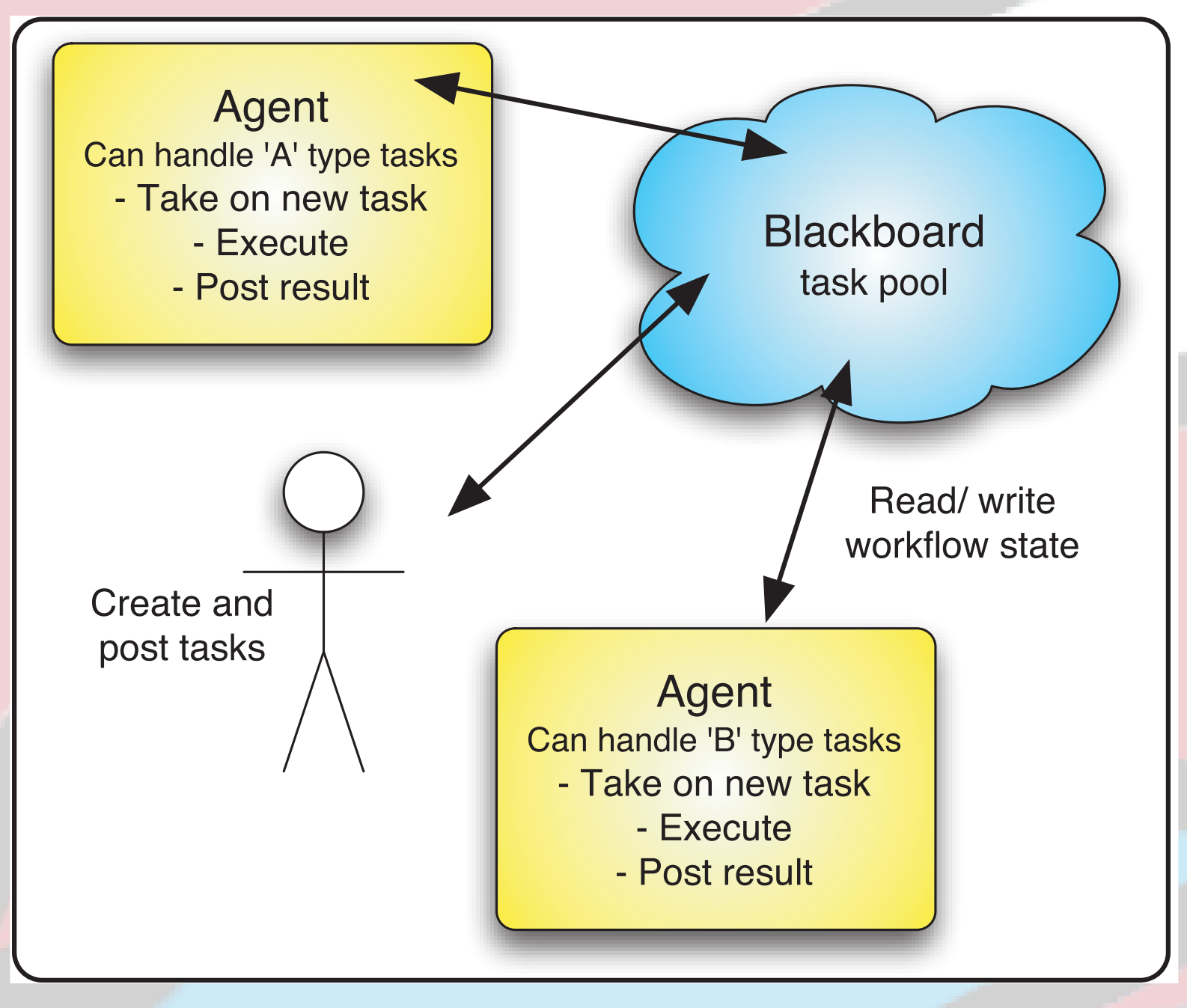
http://cern.ch/cms-project-phedex/

[1]Rehn et al, "PhEDEx high-throughput data transfer management system", CHEP' 06
[2] Barrass et al, "Software agents in data and workflow management," Computing in High Energy Physics (CHEP04), Interlaken, 2004
[3] CMS Collaboration, "The Compact Muon Solenoid Computing Technical Proposal," CERN/LHCC 1996-045 (1996).
[4] European Centre for Nuclear Research (CERN), http://www.cern.ch.
[5] "The Foundation for Intelligent Physical Agents," http://www.fipa.org.
[6] Corkill, "Collaborating Software: Blackboard and Multi-Agent Systems and the Future," Proceedings of the International Lisp Conference, New York, 2003.
[7] Gelernter, "Mirrorworlds", Oxford University Press, 1992.
[8] Anderson, Korpela, Watson, "High-Performance Task Distribution for Volunteer Computing", First IEEE International Conference on e-Science and Grid Technologies, 2005.
[9]Venema, "Postfix", http://www.postfix.org.
[10] Maymounkov and Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric", 2nd International Workshop on Peer-to-Peer Systems, 2003.
[11] Wikipedia, "Overlay network", http://en.wikipedia.org/wiki/Overlay_network.
[12] Cormen, Leiserson, Rivest and Stein, "Introduction to Algorithms", Second Edition, MIT Press and McGraw-Hill, 2001.
[13] Kyte, "tkprof", http://www.oracleutilities.com/OSUtil/tkprof.html.
[14] Navas and Wynblatt, "The network is the database: data management for highly distributed systems", Proceedings of the 2001 ACM SIGMOD international conference on Management of dat, 2001
[15] PhEDEx live monitoring, http://cms-project-phedex.web.cern.ch/cms-project-phedex/cgi-bin/browser
[16] RRD, http://people.ee.ethz.ch/~oetiker/webtools/rrdtool.
[17] MonALISA, http://monalisa.cacr.caltech.edu/monalisa.htm.
[18] IBM "Autonomic Computing", http://www.research.ibm.com/autonomic.
[19] Cozens, "Object Oriented Perl", http://www.perl.com/pub/a/2001/11/07/ooperl.html.
[20] Perl DataBase Interface, http://dbi.perl.org.
[21] Oracle, http://otn.oracle.com.
[22] Bunce, "Advanced DBI Tutorial", http://search.cpan.org/~timb/DBI_AdvancedTalk.
[23] Kyte, "Effective Oracle by Design", Osborne ORACLE press series.

Distributed workflows

Each site participating in PhEDEx transfers runs a suite of agents [5]. Each agent performs a specific small step of the workflow.

Most PhEDEx agents communicate indirectly via a blackboard [6], or tuple space[7], implemeted on a high availability database. These structures have been used to coordinate the concurrent processing of large quantities of similar data [8, and derivative projects e.g. SETI@Home]. PhEDEx however uses the blackboard to coordinate quite sophisticated workflows involving many steps.



PhEDEx architecture. Workflow tasks are created and posted on the blackboard, which acts as a task pool. Here, task B can only be undertaken once task A has been completed, so a handover of responsibility is required. In PhEDEx repsonsiblity for tasks is generally pre-allocated, although they could be picked dynamically. When a task is complete, the agent posts status to the blackboard. Often this effectively creates a new task for a another agent. The transfer workflow is defined by these state exchanges.

For complex local workflows some agents use the file system in a manner very similar to mail transport agents [9] to persist workflow state. We are currently investigating peer-to-peer agents [10] to distribute parts of the tuple space.

Agents are given increasing levels of autonomy to make decisions based on local (perceived) conditions. To limit the complexity of the system the software components are only indirectly dependent on each other-- no component knows that any other component exists, let alone what they do.

Robust agent design

PhEDEx agents do not maintain internal state. Agents can be stopped or started without ill consequences even after a system crash. Permanent workflow state is stored on the blackboard, and changes to this state are transaction-safe. Agents and sites are restricted to changing relevant partitions of the database using fine-grained database role grants, further reducing damage from operational mishaps.

Each agent is responsible for one unreliable operation or workflow step, (e.g. file transfer, checking stager status or managing the overlay network partition for a site.) Each agent:

- Finds pending work on the blackboard.
- Picks/prioritises tasks and executes them.
- Marks successful tasks complete, possibly indirectly assigning tasks for other agents.

SQL operations, embedded verbatim in agent code, are used for all communication with the blackboard. No intermediate server tiers or encapsulated SQL code in separate libraries is used.

Using a central high-availability Oracle database cluster with 24/7 support has been advantageous. The gains in system robustness, availability and flexibility have greatly outweighed the complexity of distributing the client software and other limitations.

Distributed handshakes

Tasks are joined to form a workflow using a handshake-- a writing and reading of state information to and from the blackboard by a pair of agents. The definition of what state information is available at the start of a step, and what should be available at the end, is the basis for the design of agents, which are implemented and then act only on the appearance of state information.

This translates conveniently to inserting a row into a task table in the database. To take on a task an agent simply reads a row and acts on it. When a task has been successfully completed the row is updated, and potentially new rows are inserted. In complex workflows the state is check-pointed in the database, in which case processing continues from the last good check-point on agent restart, or when the agent finds itself idle, so that they recover from long-term skew of "lost" work.

When two agents need to co-operate more closely, for example for a file download, we use a strict state machine for each database row. This enables each party to operate on several entries in identical states. In certain such state machine exchanges the agents require the other party to actively refresh the state -- if the other party is unable to regularly refresh the state, it's most likely also unable to do any useful work, and can be safely ignored.

Database access

PhEDEx is written in object-oriented perl [19] and uses the DBI [20] Oracle [21] interface. Good advice is already widely available [22, 23] and will not be repeated here. Suffice it to say the stated best practice guidelines are important to follow.

Robust common operations

We provide convenience functions for common database operations. While making the database programming easier, these functions allow us to handle problems in the distributed environment. In particular we detect stuck or bad database, and statement, handles and flag connections as unstable, enabling agents to recover and reconnect.

Maintain availability

Our procedures aim to keep the database available at all times, although it naturally some interventions are inevitable. We can remotely schedule agents to back off and to resume operations, allowing in-place tweaking without disturbing existing connections.

Significant upgrades, requiring a shutdown of ~0.25 day, are infrequent.

- Deploy and test on developer testbed.
- Develop, test, verify migration procdure on developer testbed.
- Test migration on integration testbed in collaboration with a small set of sites.

Once integration tests are complete we typically schedule 1-4 hour downtime to migrate the production instance, and resume operation immediately.

Defensive coding

Assume every operation, however trivial, will fail. We provide safe operators for numerous tasks (e.g. writing temp files; launching subprocess queues). Check for internal errors to stop errors spreading.

Failure recovery tactics

- Log the issue, back out and retry next time, or after a cool-off.
- Clear local state cache, rely on global system consistency to trigger retry.
- Flag repeated problems as bad, alert operator and ignore.

Experience shows that many commands return misleading exit codes. We treat them with scepticism; each transfer is independently cross-checked for file existence and size. Checksums can also be checked.

Simplicity

We start with simple algorithms throughout. For example, transfer failure handling has evolved from a simple retry next time, through cooling-off processes to limited queue randomisation and prioritisation. Gradually subtler tactics are being implemented where necessary, making the system increasingly more autonomous [18]. For example, some of the more advanced agents detect pathological patterns and automatically throttle themselves. Such behaviour is basically a set of higher-order corrections, suited to systems with a stable, reliable underlying fabric where response is linear.

Algorithms

The routing overlay

An overlay network is used to describe a topology in which nodes represent storage resources, independent of the underlying network fabric [11]. This allows PhEDEx to cache data at regional centres for distribution to smaller sites.

The overlay network is maintained by a link-state algorithm, with shortest paths calculated using Dijkstra's algorithm [12]. A neighbour-list containing static link-weight information is stored on the blackboard . Routing agents act at and on behalf of each node in the network, and use Dijkstra's algorithm to dynamically refresh a minimum spanning tree from their node to each other node in the network. This minimum spanning tree information is then stored in a routing table on the blackboard holding source, destination, gateway, hops information.

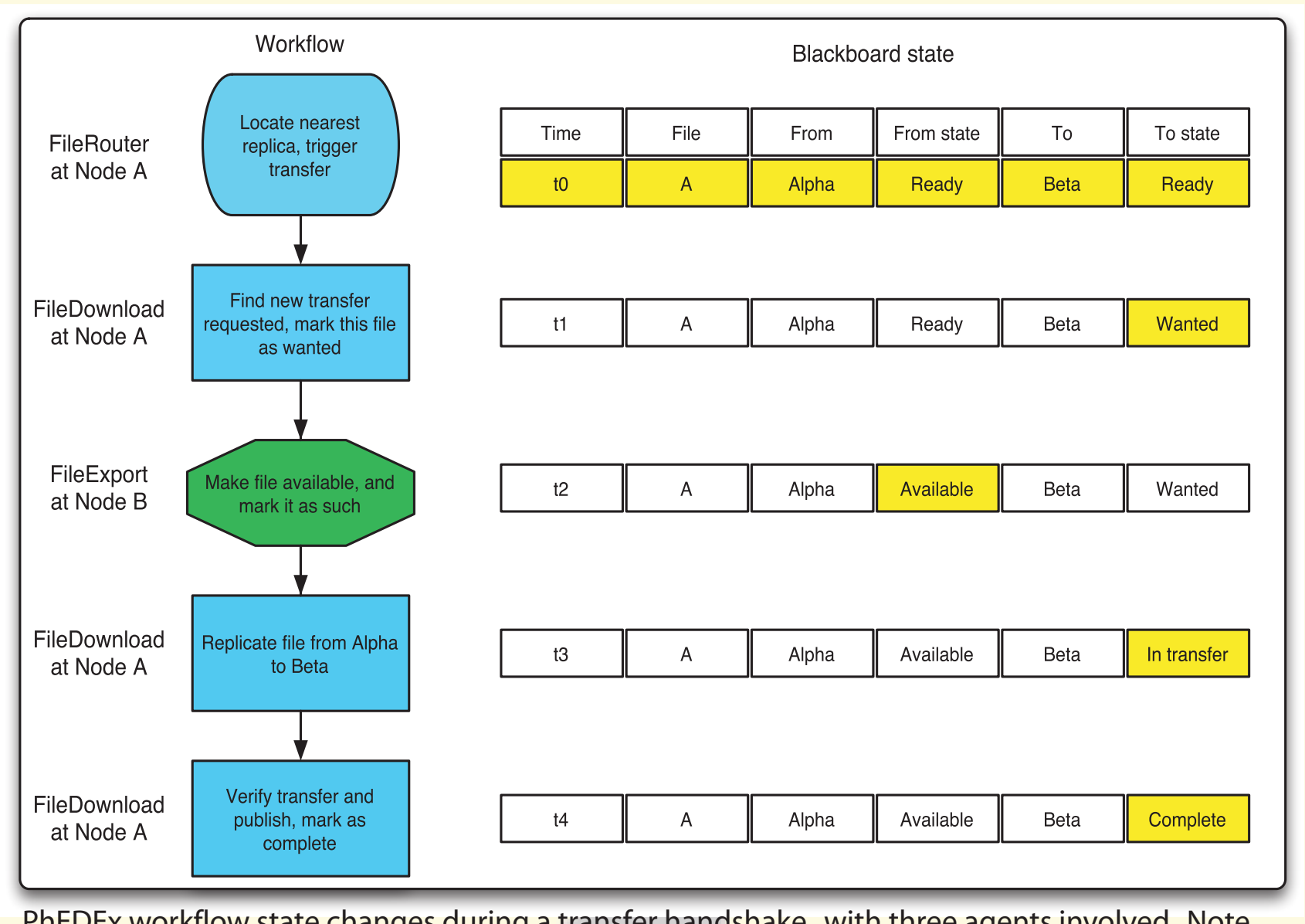
Routing files to destinations

The PhEDEx topology is a weighted, generally not fully-connected graph. Files may need to be temporarily replicated to a regional centre before final replication to a destination, to better manage the load on the central facility.

A file routing agent acts on behalf of each node in the network, and is responsible for triggering the set of replications that glue a transfer from source to destination together. The file router uses the routing table to determine shortest paths from source to destination, and triggers the first transfer in the chain by inserting a row into a transfer state table giving source and destination information. When that transfer is marked complete it reevaulates the closest replica for each file and triggers the next transfer in the chain.

Robust transfer handshake

PhEDEx developed during times of unrest in underlying storage and transfer technology. Much functionality desired of storage systems -- stage-on-demand, intelligent grouping of stage requests, sophisticated space management -- is still not in evidence. The PhEDEx transfer handshake/workflow is therefore sophisticated and incorporates much of the functionality desired of underlying systems.


Workflow
Blackboard state
FileRouter at Node A
Locate nearest replica, trigger transfer
FileDownload at Node A
Find new transfer requested, mark this file as wanted
FileExport at Node B
Make file available, and mark it as such
FileDownload at Node A
Replicate file from Alpha to Beta
FileDownload at Node A
Verify transfer and publish, mark as complete

Time	File	From	From state	To	To state
t0	A	Alpha	Ready	Beta	Ready
t1	A	Alpha	Ready	Beta	Wanted
t2	A	Alpha	Available	Beta	Wanted
t3	A	Alpha	Available	Beta	In transfer
t4	A	Alpha	Available	Beta	Complete

PhEDEx workflow state changes during a transfer handshake., with three agents involved. Note that the transfer operation is a sub-workflow, with pre-delete, bypass, transfer, verify and publish steps. The export step replicates functionality expected of underlying storage systems. The state transitions on the blackboard define the handovers of responsibility between distributed agents that together comprise the overall workflow.

```
# Pick up and process work
sub idle
{
    my ($self) = @_;
    my $dbh = undef;
    eval
    {
     my $start = &mytimeofday();
     $dbh = &connectToDatabase ($self) or die "failed to connect";

    # Find out what's posted for us to work on
     my %args = (":node" => $self->{MYNODE}, ":now" => $start);
     &dbexec($dbh, qq{
         update t_transfer_state
         set from_state = 1, from_timestamp = :now
         where to_node = :node and from_state = 0},
         %args);
     &dbexec($dbh, qq{
         update t_transfer_state
         set to_state = 2, to_timestamp = :now
         where to_node = :node and to_state = 0},
         %args);
     $dbh->commit();

     # Process files pending migration.
     my $rsstmt = $dbh->prepare(qq{
         insert into t_replica_state
         (timestamp, guid, node, state, state_timestamp)
         values (:now, :guid, :node, 0, :now)});
     my $tsstmt = $dbh->prepare(qq{
         update t_transfer_state
         set to_state = 3, to_timestamp = :now
         where guid = :guid and to_node = :node});
     my ($qstmt) = &dbexec($dbh, qq{
         select ts.guid, f.filesize, ts.from_timestamp
         from t_transfer_state ts
         join t_file f on f.guid = ts.guid
         where ts.to_node = :node and ts.to_state = 2},
         ":node" => $self->{MYNODE});

     while (my $row = $qstmt->fetchrow_arrayref())
     {
         my ($guid, $size, $time) = @$row;
         my $pfn = &guidToPFN ($guid, "srm", "local", @{$self->{PFN_QUERY}});
        my $status = 0;
        open(INFO, "srm-get-metadata $pfn |");
        while(<INFO>) {
         if (/isPermanent :true/) {
             $status = 1;
         }
        }
         close (INFO)
            or &alert( "No dCache migration info for $pfn: $!" );

         next if !$status;

         # Migrated, mark transfer completed
         &dbbindexec($tsstmt, %args, ":guid" => $guid);
         &dbbindexec($rsstmt, %args, ":guid" => $guid);
         $dbh->commit ();

         # Log delay data.
         my $nowh = &mytimeofday();
         &logmsg ("xstats: $guid $self->{MYNODE} 3 "
                . sprintf('%.2f', &mytimeofday() - $time)
                . " $size");

         # Give up if we've taken too long
         last if $nowh - $start > 10*60;
     }
    };
    do { &alert ("database error: $@");
         eval { $dbh->rollback() } if $dbh; } if $@;

    &disconnectFromDatabase ($self, $dbh);
    $self->nap ($self->{WAITTIME});
}
```

Agents derive from a base Agent class that handles initialisation tasks common to all agents. Each agent overrides 'idle', which is called iteratively when initialisation is complete.

Here the agent connects to the central blackboard. This automatically registers this agent, picks up and acts on any messages waiting for it, and opens a Perl::DBI connection to the database.

The agent acts on the blackboard , picking up any migrations (migrations and replications are both referred to as transfers) are awaiting its attention. These are indicated by from_state = 1 (available). :node represents the node at which the agent runs. These transfers are set as to_state = 2 (in transfer).

Comunication with the blackboard is not mediated by any special framework: Perl::DBI is used to connect to the database and execute SQL, and SQL embedded within the agents.

Potentially complex SQL statements are prepared in advance, before being used to act on information in the blackboard.

Bind variables are used wherever possible, especially in situations where the same query is used repeatedly.

Details about a new replica are posted on the blackboard.

Transfers are marked as complete, indicated by to_state = 3.

Transfer details are read from the blackboard so the agent can query local resources for migration state.

Read from the blackboard using the transfer details query, and process each result. Each result represents a pending transfer.

Replicas are referenced by GUID; use this GUID to lookup the physical filename of the local replica in some local resource.

Query the local resource for migration state-- the resource is still repsonsible for handling the migration operation, this agent just passivaly checks to see whether it's been done. Handle any error as a lack of information.

When the file has been migrated to tape it is treated as a new replica hosted by the node. The agent uses the prepared statements to post the details of the new replica and update the transfer state on the blackboard.

Alerts, errors and normal status output are logged to local logfile using a simple logging module.

For information, successful migrations are also logged to local logfile.

If any problem is found with communication with the blackboard, all transactions are rolled back so they can be retried.

The agent tidies up its connection to reduce load on the blackboard, and goes to sleep before trying to pick up more work.

Performance

Basic schema design and tuning

Direct access to database resources and gurus is essential for database performance. We avoid using generic services that add an extra layer of indirection and processing over those provided by the database in favour of known, manual optimisations.

Fundamental objects given primary keys	Constraints enforce data consistency at schema level, avoid error checking at client level
Foreign keys indexed, used to alias fundamental objects	
Suitable degree of denormalisation	Reduce number of joins required on hot tables
'Good' column ordering	Improves Oracle storage performance
Use bind variables in SQL	Reduce latency in query processing
Cost based query optimisation with regularly updated stats	Used to improve schema by identifying hot data and high-load queries, and inform storage decisions for administrators
Analyse important queries with oracle tools (explain plan..) and Tom Kyte's tkprof [13]	
Row movement enabled for busy tables	Improves concurrency for busy operations that require strict data consistency, and storage compaction for busy tables
Increase 'Initrans'	

Client or database processing?

We've frequently changed the definition of a problem or an algorithm such that it can be executed as a small number of SQL statements rather than client-side logic. We also:

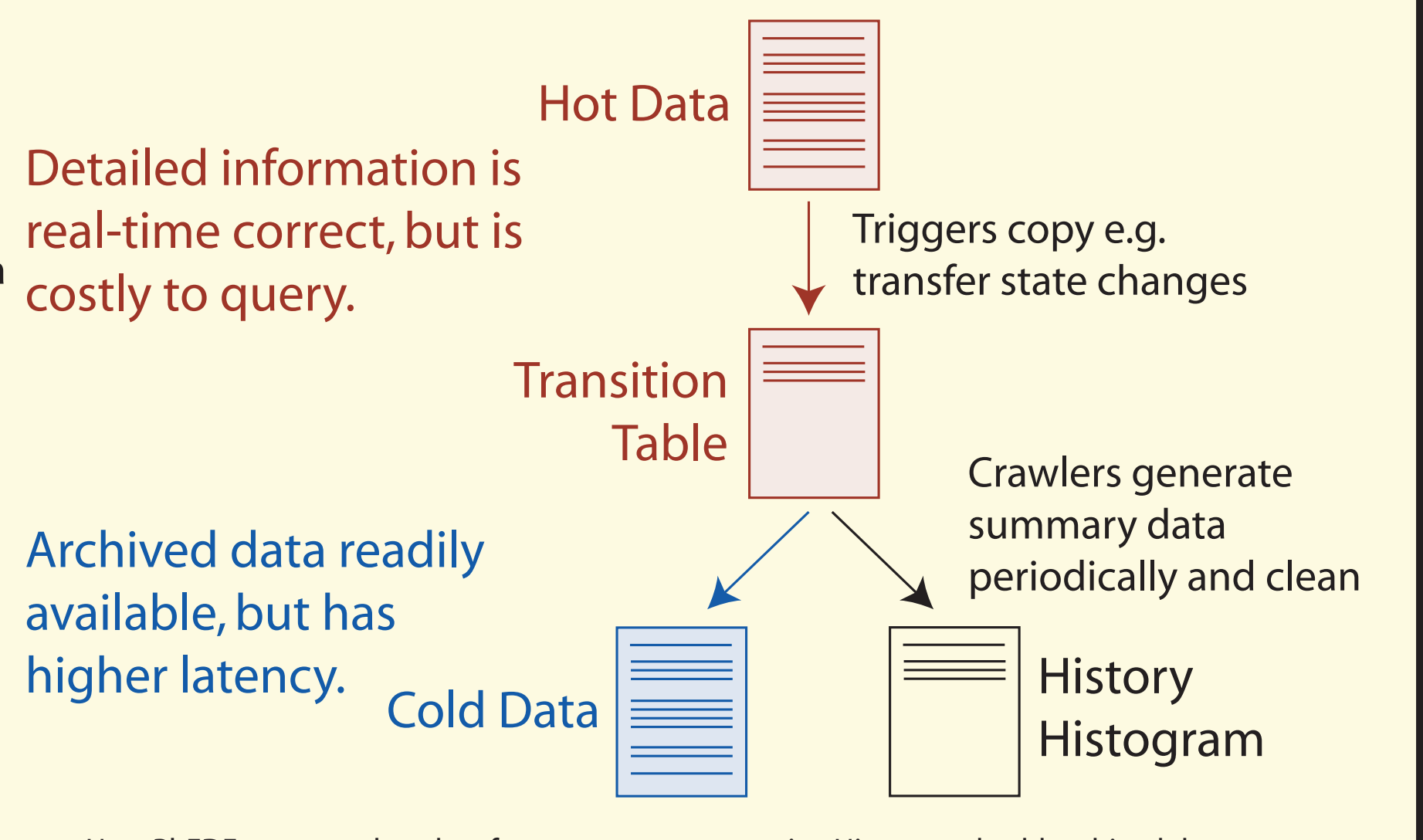
- Use no stored procedures, few triggers.
- Divide responsibility between client and datbase engine intelligently-- pulling data over WAN links is unwise, a big join is more efficient.

Exploiting data relationships

All data either has internal relationships allowing it to be grouped hierarchically, or can have arbitrary relationships imposed upon it. PhEDEx divides whole of the data into "streams", those further into "blocks", which contain files. Operating on streams and even on blocks is extremely fast, since the tables describing them are compact.

Moreover, by operating on only "active" blocks-- by expanding file information into hot tables only when they are actively being transferred-- we massively reduce the number of operations that touch file-level data at any time.

Separating hot, active data


Hot Data
Detailed information is real-time correct, but is costly to query.
Triggers copy e.g. transfer state changes
Transition Table
Crawlers generate summary data periodically and clean
Archived data readily available, but has higher latency.
Cold Data
History Histogram

How PhEDEx protects hot data from unnecessary queries. History and cold archived data are updated serially, in a single operation. Histograms are filled first, then the data copied toarchive tables; remnant data is removed from the transition table. During this period the transition table is locked so that it doesn't get out of sync.

Smart caching in stateless agents

Although the agents are stateless with regard to critical global workflow state, some agents build caches to improve throughput. Caches tag data with a validity of some hours after which the record is purged and reloaded from the database on next use. This makes the agents self-healing. Caches are used only when the agent is the sole authoritative source for the data so it only needs to shield itself against direct database changes, not changes by other agents. One example is the agent managing the stage space of a given tape system at a site.

High performance monitoring

Web monitoring pages showing current live state [15] can inhibit the performance of sizeable and active databases, let alone presenting historical plots and statistics.

- Auxiliary monitoring tables are filled by background processes at regular intervals; web pages query these tables.
- Update frequency depends on source data; most just captures overall state with a large 'select ... group by ...'.
- Compromise between query cost and user requirements for observation-- varies between 40 s and 15 min.
- Visible updates guaranteed every 4-5 minutes using multiple layers of aggregation, independent of database load.

- Fine-grained partial histograms with 5-10 min bins represent historical data (e.g. see RRD [16] and MonALISA [17].)
- Histograms updated on movement from hot to cold tables via holding tables.
- Guarantee we never need to query full historical data to build accurate, low latency summaries. Query time depends only on number of time bins queried, not table load.

- Agents can access this historical information to help adjust their own behaviour, bringing us closer to developing an autonomic system [18]. Adding dynamic behaviour is, however, complex.