CERN
Tape Archive

Archiving, Reporting and DB locking
(Scheduler DB)

Jaroslav Guenther (IT-SD-TAB)

31/S-028

04.07.2024

Local test run setup

Setup with Postgres Scheduler DB

- dev VM **2xlarge** (16 VCPU, ~28GB RAM)
- catalogue Postgres **container**
- scheduler Postgres DBOD (dbod-pgscheddb.cern.ch)
- CI containerised deployment (tpsrv01, tpsrv02, ctaeos, ...)

CI Setup

- CI VM **xlarge** (8 VCPU, ~14GB RAM)
- catalogue Oracle (devdbs2-rac16-scan.cern.ch:10121/castorint.cern.ch)
- scheduler Objectstore (cephkelly.cern.ch)
- CI containerised deployment (tpsrv01, tpsrv02, ctaeos, ...)

Local test run comparison

Archive Workflow (think 1 DB table)

- **updating set of rows to take ownership**
  - rows locked only for write (default UPDATE lock used only)
  - when mount picks new set of jobs
  - when DiskReportRunner picks new set of jobs
- **incomplete reporting**
  - only successful jobs reported, crash otherwise
- 10 000 files in 1 directory each 15kB (client_archive.sh)

(creation_time - last_update_time)
~ 0.5 ± 0.3 sec/job

```
Starting at 1720159352
Copying files to /eos/ctaeos/preprod/4d8c60e8-7c1d-4166-9446-4a03feff43cd/0 using 100 processes...
10000/10000 archived checked within 2 seconds, current timestamp 1720159456
```

comparable overall throughput
~ 100 ± 10 Hz

Objectstore backend

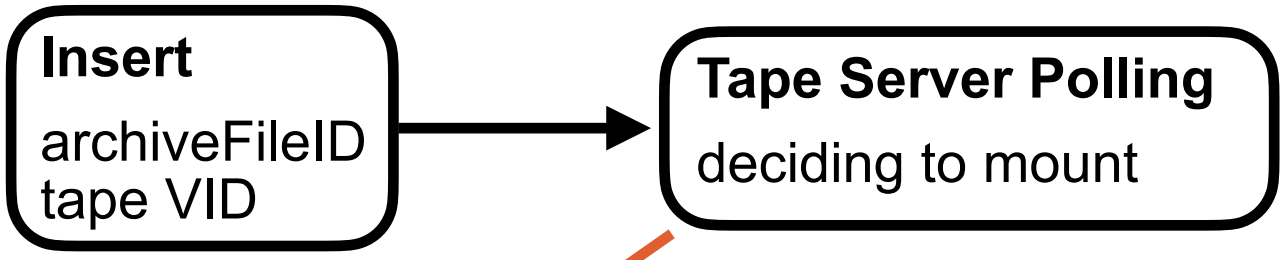
- same test

```
Starting at 1720138107
Copying files to /eos/ctaeos/preprod/87c2cb38-2ba7-4edc-ae01-6c1d9e01fa75/0 using 100 processes...
10000/10000 archived checked within 3 seconds, current timestamp 1720138205
```

Archiving workflow

... step by step !

Archiving workflow - insert & mount poll



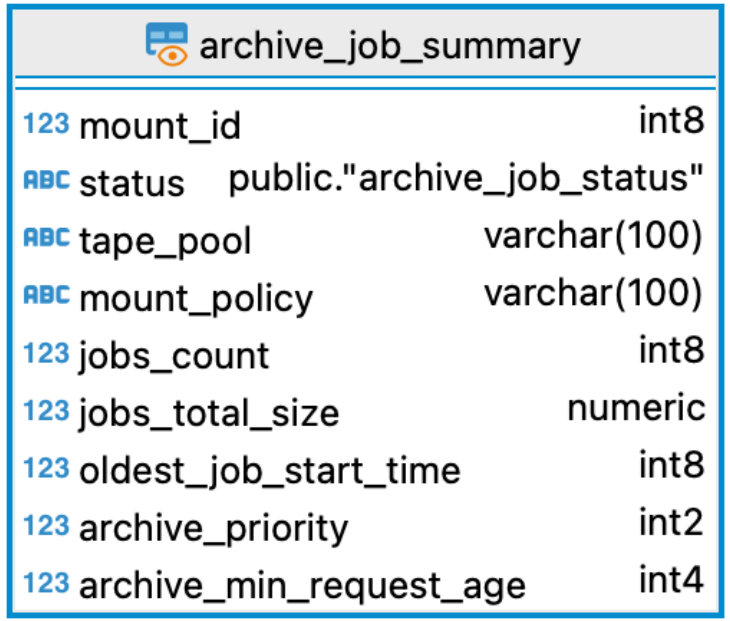
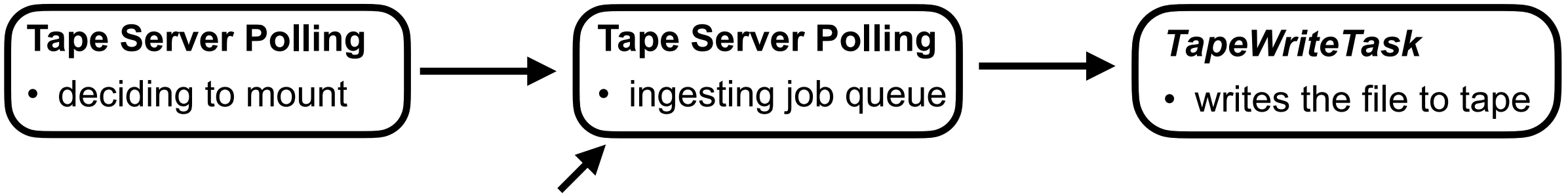


Table View

- just a query saved query
- **executed every poll !**
- expensive for large tables
- optimisation options
  - ➡ PG has **materialised view**
    - refresh by query every 30 sec
    - refresh by pg_cron extension
  - ➡ custom counter implementation (insert - pop & xcheck)

```
CREATE VIEW ARCHIVE_JOB_SUMMARY(
  MOUNT_ID,
  STATUS,
  TAPE_POOL,
  MOUNT_POLICY,
  JOBS_COUNT,
  JOBS_TOTAL_SIZE,
  OLDEST_JOB_START_TIME,
  ARCHIVE_PRIORITY,
  ARCHIVE_MIN_REQUEST_AGE) AS SELECT
    MOUNT_ID,
    STATUS,
    TAPE_POOL,
    MOUNT_POLICY,
    COUNT(*),
    SUM(SIZE_IN_BYTES),
    MIN(START_TIME),
    MAX(PRIORITY),
    MIN(MIN_ARCHIVE_REQUEST_AGE)
FROM ARCHIVE_JOB_QUEUE GROUP BY
    MOUNT_ID,
    STATUS,
    TAPE_POOL,
    MOUNT_POLICY;
```

Archiving workflow - file transfer




```
std::string sql =
  "WITH SET_SELECTION AS ( "
    "SELECT JOB_ID FROM ARCHIVE_JOB_QUEUE "
  "WHERE TAPE_POOL = :TAPE_POOL "
  "AND STATUS = :STATUS "
  "AND MOUNT_ID IS NULL "
  "ORDER BY PRIORITY DESC, JOB_ID "
  "LIMIT :LIMIT FOR UPDATE) "
  "UPDATE ARCHIVE_JOB_QUEUE SET "
    "MOUNT_ID = :MOUNT_ID,"
    "VID = :VID "
  "FROM SET_SELECTION "
  "WHERE ARCHIVE_JOB_QUEUE.JOB_ID = SET_SELECTION.JOB_ID "
  "RETURNING SET_SELECTION.JOB_ID";
```

Job selection for processing

- update first 500 jobs
  - where MOUNT_ID IS NULL (aka not owned)
  - status == 'AJS_ToTransferForUser'
- updates mount_ID and VID
- then use the JOB_IDs returned to run a separate SELECT to get all the job info (query not shown here)

italics = thread

Archiving workflow - reporting

TapeWriteTask

- at successful write
- reports job to ***MigrationReportPacker***
  - **tape file***: size, checksum, ...*

MigrationReportPacker thread

- collects all reports
- calls ArchiveMount::**reportJobsBatchTransferred**()
  - **validates the success**
    - checks disk/tape checksum, file size etc.
    - if OK calls ArchiveMount::**setJobBatchTransferred** ()
    - and 'SchedulerDB'::**updateJobStatus**()

italics = thread

Archiving workflow - reporting

MigrationReportPacker

- 'SchedulerDB'::updateJobStatus()

```
:STATUS ='AJS_ToReportToUserForTransfer'
```

```
for (const auto &piece : jobIDs) sqlpart += piece + ",";
if (!sqlpart.empty()) { sqlpart.pop_back(); }
std::string sql =
        "UPDATE ARCHIVE_JOB_QUEUE SET "
        "STATUS = :STATUS "
        "WHERE JOB_ID IN (" + sqlpart + ") ";
```

Disk*ReportRunner* - *now at any tape server*

- collects all reports (failed and successful)
- for successes picks up the jobs with status 'AJS_ToReportToUserForTransfer'

italics = thread

Archiving workflow - reporting

Disk*ReportRunner*
now at any tape server

- mark jobs with 'AJS_ToReportToUserForTransfer' status as IS_REPORTING and returning job IDs
- SELECT job info by jobID list

- then use the JOB_IDs returned to run a separate SELECT to get all the job info (query not shown here)

- calls Scheduler.reportArchiveJobsBatch()
  - **reports status back to EOS**
  - setArchvieJobBatchReported()

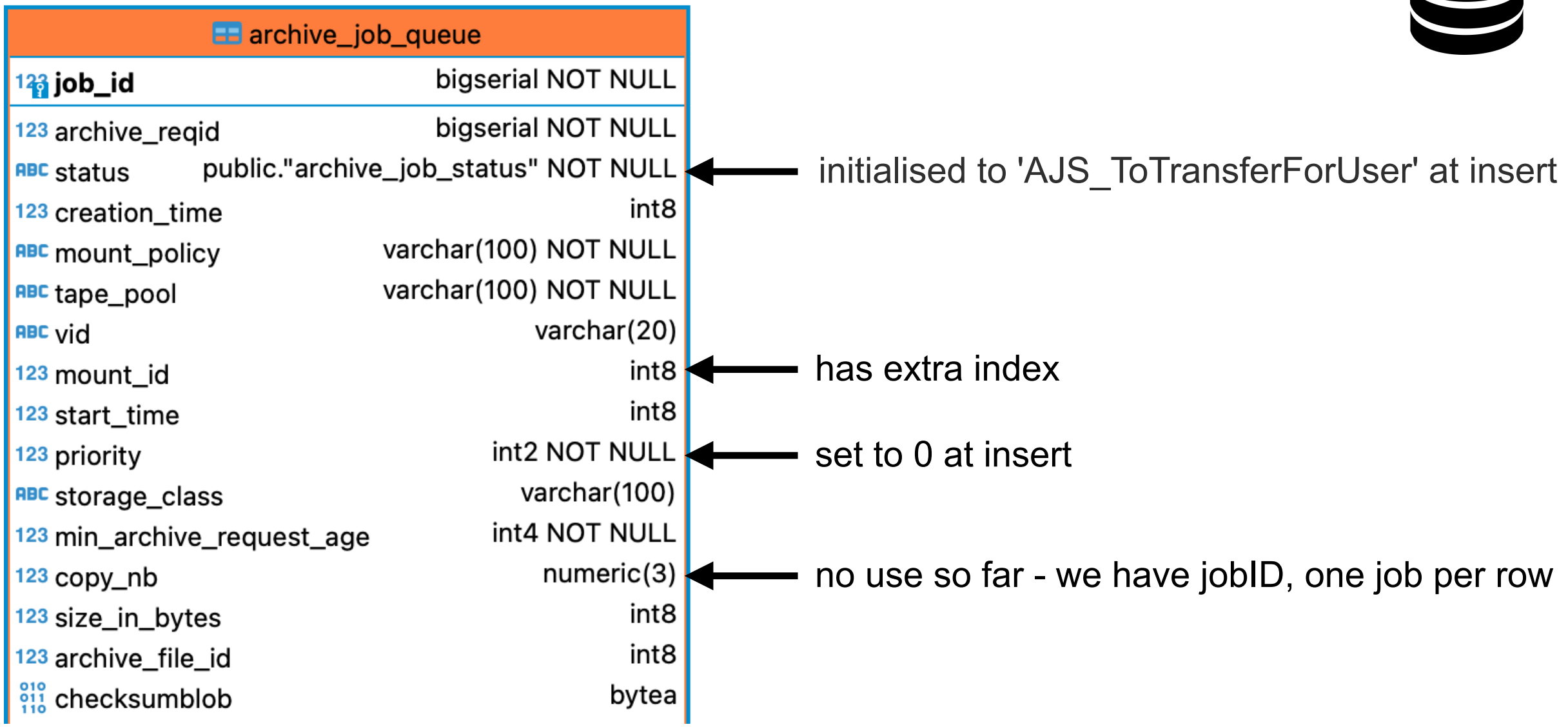
```
"WITH SET_SELECTION AS ( "
"SELECT JOB_ID FROM ARCHIVE_JOB_QUEUE "
"WHERE STATUS = ANY(ARRAY['AJS_ToReportToUserForTransfer', "
                         'AJS_ToReportToUserForFailure']";
"]::ARCHIVE_JOB_STATUS[]) AND IS_REPORTING IS NULL)  "
"ORDER BY PRIORITY DESC, JOB_ID  "
"LIMIT :LIMIT FOR UPDATE) "
"UPDATE ARCHIVE_JOB_QUEUE SET "
"IS_REPORTING = 1 "
"FROM SET_SELECTION "
"WHERE ARCHIVE_JOB_QUEUE.JOB_ID = SET_SELECTION.JOB_ID "
"RETURNING SET_SELECTION.JOB_ID";
```

```
for (const auto &piece : jobIDs) sqlpart += piece + ",";
if (!sqlpart.empty()) { sqlpart.pop_back(); }
std::string sql =
        "UPDATE ARCHIVE_JOB_QUEUE SET "
        "STATUS = :STATUS "
        "WHERE JOB_ID IN (" + sqlpart + ") ";
```

```
:STATUS ='AJS_Completed'
```

Archive job queue schema

1/3



Archive job queue schema

2/3

ABC requester_name	varchar(100)
ABC requester_group	varchar(100)
ABC src_url	varchar(2000)
ABC disk_instance	varchar(100)
ABC disk_file_path	varchar(2000)
ABC disk_file_id	varchar(100)
123 disk_file_gid	int4
123 disk_file_owner_uid	int4
ABC archive_error_report_url	varchar(2000)
ABC archive_report_url	varchar(2000)
123 total_retries	int2
123 max_total_retries	int2
123 retries_within_mount	int2
123 max_retries_within_mount	int2
123 last_mount_with_failure	int8
ABC is_reporting	bpchar(1)
last_update_time	timestamp

to be understood once
I start dealing with failed jobs
and failed reports

is_reporting - marks reports taken
by DiskReportRunner

added since it is very useful for debugging
and might be useful dealing with dangling
owned jobs without active mounts

Archive job report - placeholder

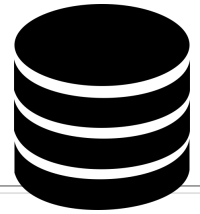
archive_job_reports	
job_id	bigserial NOT NULL
status	public."archive_job_status" NOT NULL
creation_time	int8
mount_id	int8
start_time	int8
priority	int2 NOT NULL
storage_class	varchar(100)
copy_nb	numeric(3)
size_in_bytes	int8
archive_file_id	int8
checksumblob	bytea
requester_name	varchar(100)
requester_group	varchar(100)
disk_instance	varchar(100)
disk_file_path	varchar(2000)
archive_error_report_url	varchar(2000)
archive_report_url	varchar(2000)
failure_report_log	text
failure_log	text
total_retries	int2
max_total_retries	int2
retries_within_mount	int2
max_retries_within_mount	int2
last_mount_with_failure	int8
tape_pool	varchar(100) NOT NULL
repack_filebuf_url	varchar(2000)
repack_fseq	numeric(20)
total_report_retries	int2
max_report_retries	int2

ARCHIVE_JOB_REPORTS

- contains all other fields currently not of use for the archiving workflow, but found in Objectstore implementation
- might be useful to avoid locking the ARCHIVE_JOB_QUEUE
  - but if we lock rows we might not need it ! --> see locking further

Archive job queue schema

3/3



archive_job_queue | Enter a SQL expression to filter results (use Ctrl+Space)

	job_id	archive_reqid	status	creation_time	mount_policy	tape_pool	vid	mount_id	start_time	priority	storage_class
1	1	1	AJS_Complete	1,719,963,020	ctasystest	ctasystest	V00101	1	1,719,963,020	1	ctaStorageClass
2	2	2	AJS_Complete	1,719,963,020	ctasystest	ctasystest	V00101	1	1,719,963,020	1	ctaStorageClass
3	3	3	AJS_Complete	1,719,963,020	ctasystest	ctasystest	V00101	1	1,719,963,020	1	ctaStorageClass
4	4	4	AJS_Complete	1,719,963,020	ctasystest	ctasystest	V00101	1	1,719,963,020	1	ctaStorageClass
5	5	5	AJS_Complete	1,719,963,020	ctasystest	ctasystest	V00101	1	1,719,963,020	1	ctaStorageClass

	min_archive_request_age	copy_nb	size_in_bytes	archive_file_id	checksumblob	requester_name	requester_group	src_url
1	1	1	15,360	4,294,967,316	¯ì	user1	eosusers	root://ctaeos.nsdev.svc.cluster.local/
2	1	1	15,360	4,294,967,298	÷ Â«	user1	eosusers	root://ctaeos.nsdev.svc.cluster.local/
3	1	1	15,360	4,294,967,317	T ¿¸	user1	eosusers	root://ctaeos.nsdev.svc.cluster.local/
4	1	1	15,360	4,294,967,297	^ v	user1	eosusers	root://ctaeos.nsdev.svc.cluster.local/
5	1	1	15,360	4,294,967,296	° Bà	user1	eosusers	root://ctaeos.nsdev.svc.cluster.local/

	disk_instance	disk_file_path	disk_file_id	disk_file_gid	disk_file_owner_uid	archive_error_report_url	archive_report_url
1	ctaeos	/eos/ctaeos/preprod/7d32e2b6-d43	15	1,100	11,001	eosQuery://ctaeos.nsdev.svc.cluster.	eosQuery://ctaeos.nsdev.svc.cluster.
2	ctaeos	/eos/ctaeos/preprod/7d32e2b6-d43	20	1,100	11,001	eosQuery://ctaeos.nsdev.svc.cluster.	eosQuery://ctaeos.nsdev.svc.cluster.
3	ctaeos	/eos/ctaeos/preprod/7d32e2b6-d43	21	1,100	11,001	eosQuery://ctaeos.nsdev.svc.cluster.	eosQuery://ctaeos.nsdev.svc.cluster.
4	ctaeos	/eos/ctaeos/preprod/7d32e2b6-d43	27	1,100	11,001	eosQuery://ctaeos.nsdev.svc.cluster.	eosQuery://ctaeos.nsdev.svc.cluster.
5	ctaeos	/eos/ctaeos/preprod/7d32e2b6-d43	14	1,100	11,001	eosQuery://ctaeos.nsdev.svc.cluster.	eosQuery://ctaeos.nsdev.svc.cluster.

	total_retries	max_total_retries	retries_within_mount	max_retries_within_mount	last_mount_with_failure	is_reportdecided	last_update_time
1	0	2	0	2	0	[NULL]	2024-07-03 01:30:20.425
2	0	2	0	2	0	[NULL]	2024-07-03 01:30:20.449
3	0	2	0	2	0	[NULL]	2024-07-03 01:30:20.449
4	0	2	0	2	0	[NULL]	2024-07-03 01:30:20.452
5	0	2	0	2	0	[NULL]	2024-07-03 01:30:20.454

Locking Optimisations

Update use-cases for Locking

Archive queue ingestion:

- mount ownership: mount_id, vid updates

MigrationReportPacker:

- finished transfer, ready for report

DiskReportRunner:

- tape server report ownership (could be replaced by new insert)
- completed report update

Options for avoiding interference by:

- new inserts to another table
- ensure deterministic select statements
- all access table locks
- row write lock
  - could work fine if we do not mind duplicate state updates second tape server updates still all initially selected rows !
  - or + set stricter transaction isolation preventing read
  - but then we have error thrown (often) on any conflict & retry foreseen !
- advisory lock + smart logic on the side of the code
- …

PostgreSQL Locking

MVCC Transaction Isolation Levels:

- **Read committed (default)**
  - e.g. SELECTs within same txn can see different data from different commits (but only committed)
- **Repeatable read**
  - e.g. SELECTs within same txn see only the same data (freeze at txn start)
- **Serialisable**
  - emulates serial transaction execution for all committed transactions
- in case of conflict → ERROR
  - **txn must be retried**

Explicit Locking:
(where MVCC is not enough)

- session or transaction level
- **Advisory locks**
  - our software controls the concurrency
- **Table/Row level locks**
- **Deadlocks**
  - automatically resolved by aborting one of the transactions

PostgreSQL MVCC

txn can read **uncommitted** data

txn re-reads **row data** with different result

txn re-executes **row search** with different result

result of set of txn commits **inconsistent with any order if ran one at a time**

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

default - no reason to change for the moment

possible in PG

PostgreSQL Explicit Advisory Locks

Advisory Locks

- Transaction or Session level
  - we control the concurrency from the code by the assigned/requested lock ID

Session 1:

```
admin=> -- Acquires a lock ID 321
admin=> SELECT pg_advisory_lock(321);
 pg_advisory_lock
------------------
(1 row)
admin=> INSERT INTO TAPE_MOUNTS (MOUNT_ID)
VALUES (1);
INSERT 0 1
admin=> SELECT pg_advisory_unlock(321);
 pg_advisory_unlock
--------------------
 t
(1 row)
```

Session 2:

```
admin=> -- Attempt to acquire lock ID 321
(will block until Session 1 releases it)
admin=> SELECT pg_advisory_lock(321);
SELECT * FROM TAPE_MOUNTS;
[-- waiting for Session 1 to release it --]

[-- Session 1 released the lock 321 --]
 pg_advisory_lock
------------------
(1 row)
admin=> SELECT * FROM TAPE_MOUNTS;
 mount_id |          creation_time           |
----------+----------------------------------|
        1 | 2024-06-18 10:25:47.448468+02 |
(1 row)
```

PostgreSQL Explicit Table Locks

Table Locks

- each query will have **automatic lock** with minimal protection provided
- we can explicitly acquire locks in addition
- throw error in case of conflict

- **Session 1:**

```
admin=> BEGIN;
BEGIN
admin=*> LOCK TABLE TAPE_MOUNTS
IN ACCESS EXCLUSIVE MODE;
LOCK TABLE
[.. WORK BEING DONE ... ]
admin=*> COMMIT;
COMMIT
admin=>
```

Session 2:

```
admin=> select * from tape_mounts;

[-- waiting for Session 1 to commit --]

 mount_id | owner
----------+---------
        1 | me
(1 row)

admin=>
```

- **other modes:**
  - ACCESS SHARE = SELECT lock, only reads allowed
  - ...

PostgreSQL Row Explicit Locks

Row Locks

- FOR UPDATE blocks other writers, not readers = default for UPDATE operation !
- to block readers we may add SERIALIZABLE or REPEATABLE READ transaction isolation

Session 1:

```
admin=> BEGIN;
BEGIN
admin=*> SELECT * from tape_mounts where
mount_id=1 FOR UPDATE;
 mount_id |  owner
----------+---------
        1 |  me
(1 row)

admin=*> UPDATE tape_mounts set mount_id=2
where mount_id=1;
UPDATE 1

admin=*> COMMIT;
COMMIT
```

Session 2:

```
admin=> BEGIN;
BEGIN
admin=*>  SELECT * from tape_mounts
where mount_id=2 FOR UPDATE;
 mount_id | owner
----------+-------
        2 | you
(1 row)
admin=*> SELECT * from tape_mounts
where mount_id=1 FOR UPDATE;

[-- waiting for Session 1 to commit --]

 mount_id | owner
----------+-------
(0 rows)
admin=>
```

Backup

Postgres DB Management Challenges

MVCC (Multi-Version Concurrency Control)

- consistent "snapshot" views
- keeps all row versions until the oldest active transaction or next automatic vacuuming

Power cut & Recovery

- Incomplete transactions and vacuuming may cause long lockdowns (~1 hour) to replay WAL
- risk of data inconsistency or corruption, prevention

DBOD

- ideal for performance testing, realistic latency (RTT)
- ensure SSDs are used to avoid random access issues

& beware of implicit transactions without auto-commit keeping all history !

```
PostgreSQL config options:

---------
Write-Ahead Log (WAL) settings

wal_level = replica
synchronous_commit = on
wal_sync_method = fsync
---------
Checkpoints

checkpoint_timeout = 5min
checkpoint_completion_target = 0.7
max_wal_size = 1GB
---------
Point-in-Time Recovery

archive_mode = on
archive_command = 'cp %p /path/to/archive/%f'
---------
Streaming Replication

wal_level = replica
max_wal_senders = 5
wal_keep_segments = 32
---------
Autovacuum  (for MVCC cleanup)

autovacuum = on
autovacuum_naptime = 1min
autovacuum_vacuum_threshold = 50
autovacuum_analyze_threshold = 50

... etc.
```

Management of Completed Job Records

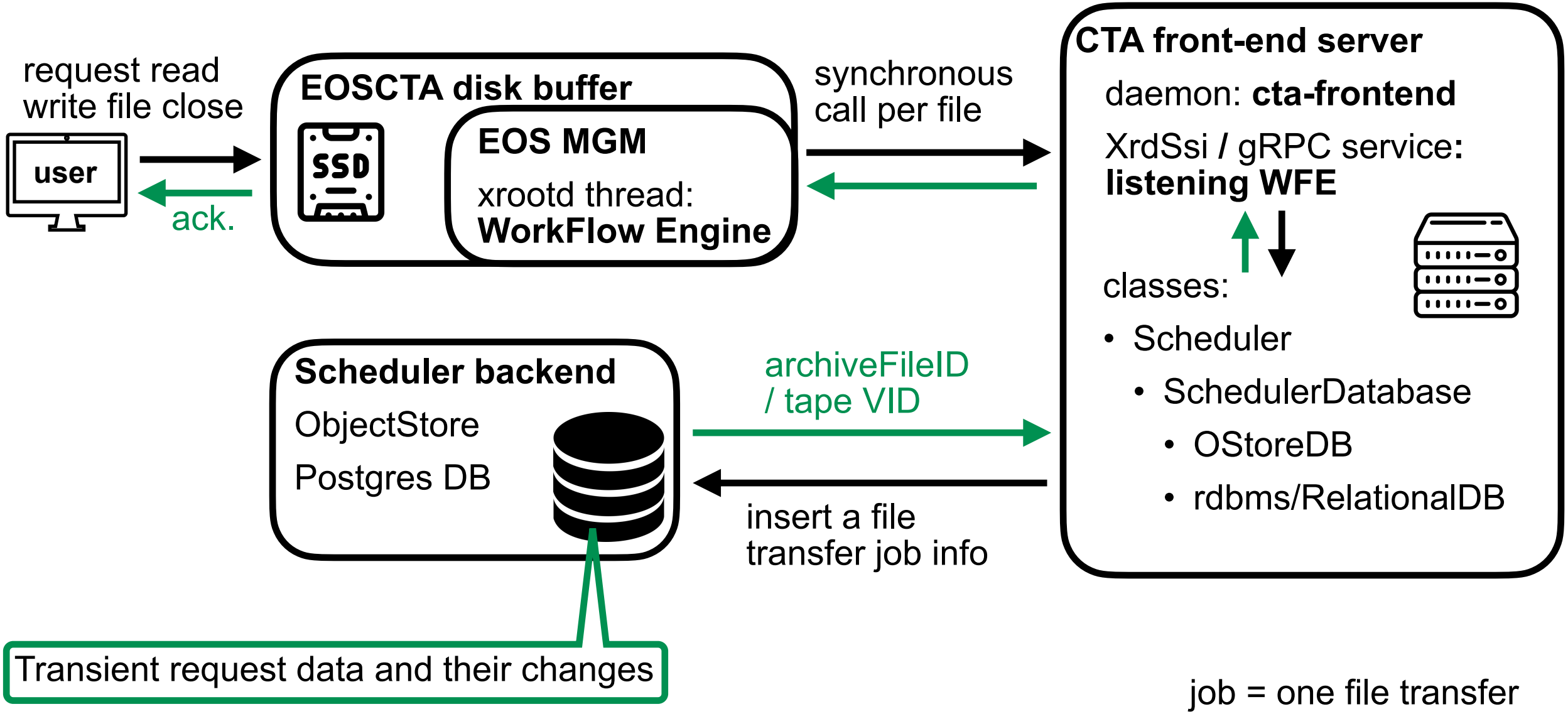
Vacuuming

- table scan + version replay + row deletion + reindexing
- **gradually reclaims disk space**
- **slower**, can lock large tables (especially: VACUUM FULL)

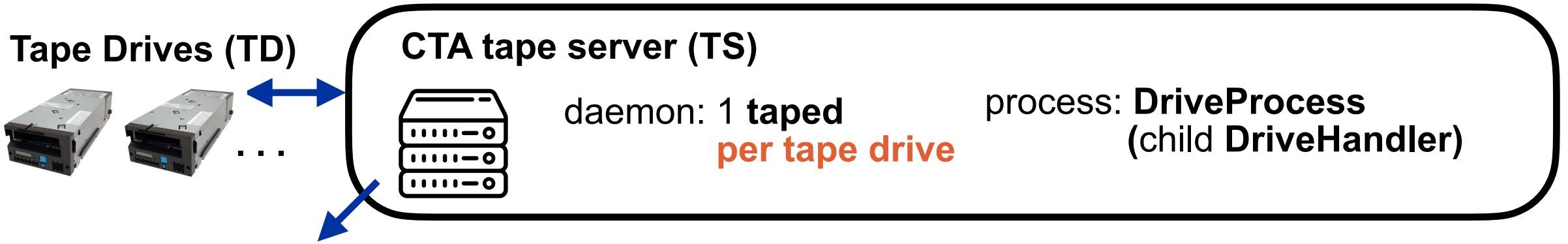
Double Buffering + Truncate Table

- use two identical tables, switch between them
- avoids extended lock periods during maintenance
- consistent data access
- Truncate obsolete table:
  - no table scan or history replay checks
  - **fast** row removal
  - **immediately reclaims disk space**

CTA Receiving a Request



CTA Tape Server polling 1/2



DataTransferSession forked for a free drive (UP)

- tries to get new (/its own) Mount
- **Mount** = drive assignment to tape for set of jobs
- calls **Scheduler → getNextMount[-DryRun]()**
  - **SchedulerDatabase → fetchMountInfo()**

improve perf **if needed**

- look up only TD relevant info
- lock only what needs to be locked

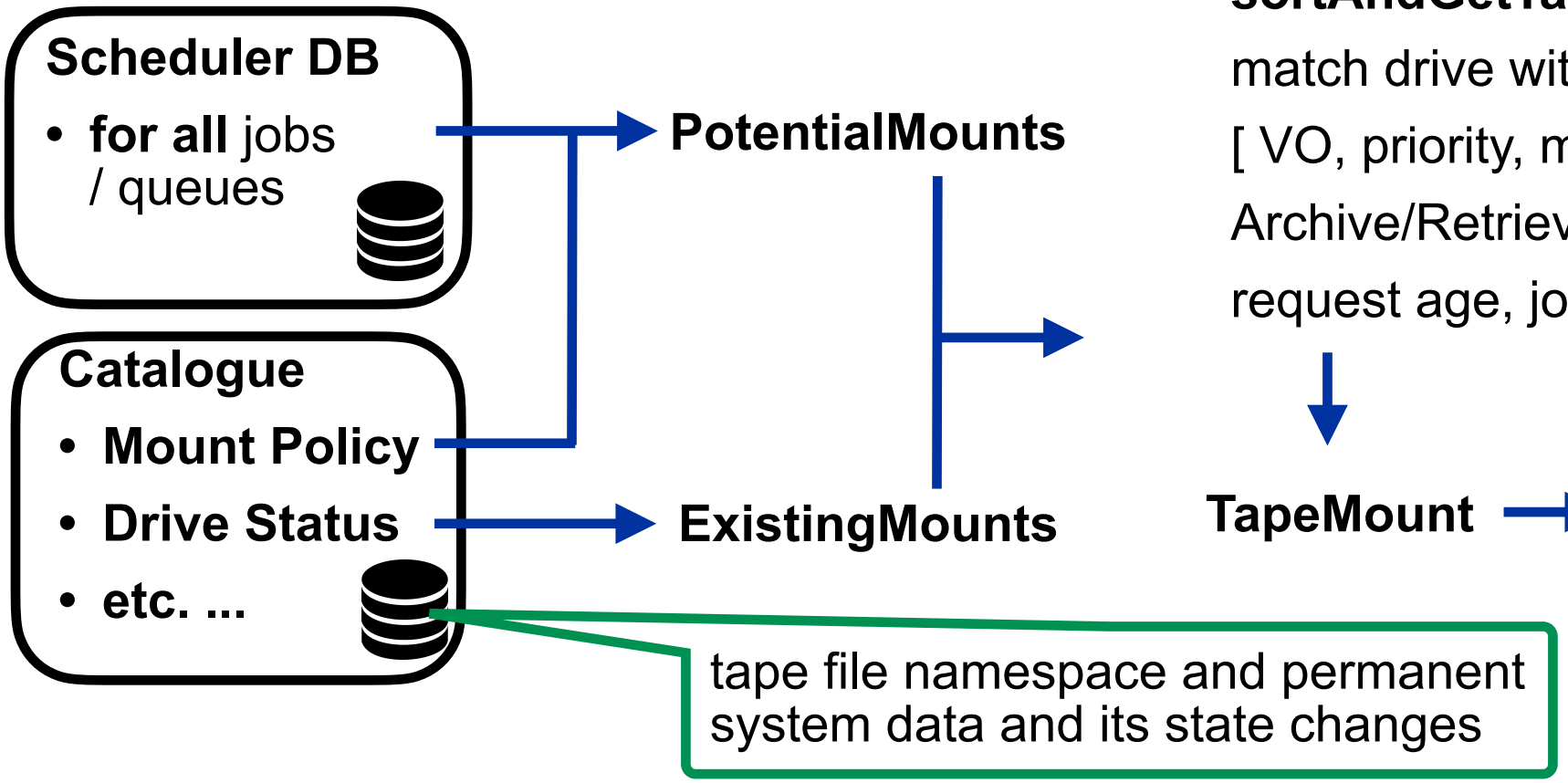
Each *taped* looks at **all jobs** for **all drives**
to get **all** (existing/hypothetical) **Mounts**
+ iterates through → match drive with tape and job set
(1st w/o global lock [-DryRun] + 2nd time with)

job = one file transfer

CTA Tape Server polling 2/2

DataTransferSession

- getting Mounts by polling Scheduler DB and Catalogue
  - **Scheduler → getNextMount[-DryRun]()**



Scheduler.
sortAndGetTapesForMountInfo()

match drive with

[VO, priority, mount type

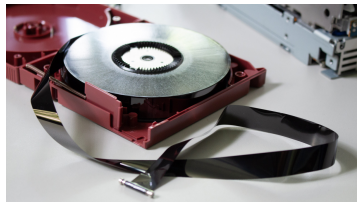
Archive/Retrieve/..., tape VID,

request age, job summary statistics]

TapeMount → **SchedulerDatabase**

updates
mountID, VID per job

CTA Tape Drive with Mount



DataTransferSession

- calls **executeWrite/Read(TapeMount)**
  - several threads are spawned taking care of:
    - **mounting** the **tape**
    - **polling Scheduler DB** for job/queue batches
    - **inserting** the **jobs** to for the execution
    - the **R/W** from/to memory/tape/disk buffer
    - **MigrationReportPacker** thread reporting back to CTA disk buffer (EOS) **(TBD for PGSCHED)**

threads handle
work of any tape drive

aka object ownership
concept in ObjectStore

Consistency & Error Handling (TBD for PGSCHED)

- TapeDaemon/**MaintenanceHandler** x-check job "heartbeat" in Scheduler DB
- Scheduler DB "view" on active [VID + mountID] → DriveState check (in the Catalogue) ?

CTA Scheduler Relational DB

Implementation

- **workflow oriented** tables, views, sequences
  - file transfer **jobs** (Archive/Retrieve/Report/...)
- inherently **uses DB features**
  - facilitates **any job ordering** (FIFO/non-FIFO) locking & MVCC, indexing (+sync), B trees, etc.
- **connection pools** from our *rdbms* wrapper layer
- currently **single threaded interface to DB**

cta_scheduler

schema_version_major	numeric(20) NOT NULL
schema_version_minor	numeric(20) NOT NULL
next_schema_version_major	numeric(20)
next_schema_version_minor	numeric(20)
status	varchar(100)
is_production	bpchar(1) NOT NULL

archive_job_summary

mount_id	int8
status	public.archive_job_status
tape_pool	varchar(100)
mount_policy	varchar(100)
jobs_count	int8
jobs_total_size	numeric
oldest_job_start_time	int8
archive_priority	int2
archive_min_request_age	int4

archive_job_queue

job_id	bigserial NOT NULL
archive_reqid	bigserial NOT NULL
status	public.archive_job_status NOT NULL
creation_time	int8
mount_policy	varchar(100) NOT NULL
vid	varchar(20)
mount_id	int8
start_time	int8
priority	int2 NOT NULL
storage_class	varchar(100)
min_archive_request_age	int4 NOT NULL
copy_nb	numeric(3)
size_in_bytes	int8
archive_file_id	int8
checksumblob	bytea
requester_name	varchar(100)
requester_group	varchar(100)
src_url	varchar(2000)
disk_instance	varchar(100)
disk_file_path	varchar(2000)
disk_file_id	varchar(100)
disk_file_gid	int4
disk_file_owner_uid	int4
archive_error_report_url	varchar(2000)
archive_report_url	varchar(2000)
failure_report_log	text
failure_log	text
is_reportdecided	bpchar(1)
total_retries	int2
max_total_retries	int2
retries_within_mount	int2
max_retries_within_mount	int2
last_mount_with_failure	int8
total_report_retries	int2
max_report_retries	int2
tape_pool	varchar(100) NOT NULL

archive_job_reports

job_id	bigserial NOT NULL
status	public.archive_job_status NOT NULL
creation_time	int8
mount_id	int8
start_time	int8
priority	int2 NOT NULL
storage_class	varchar(100)
copy_nb	numeric(3)
size_in_bytes	int8
archive_file_id	int8
checksumblob	bytea
requester_name	varchar(100)
requester_group	varchar(100)
disk_instance	varchar(100)
disk_file_path	varchar(2000)
archive_error_report_url	varchar(2000)
archive_report_url	varchar(2000)
failure_report_log	text
failure_log	text
is_reportdecided	bpchar(1)
total_retries	int2
max_total_retries	int2
retries_within_mount	int2
max_retries_within_mount	int2
last_mount_with_failure	int8
total_report_retries	int2
max_report_retries	int2
tape_pool	varchar(100) NOT NULL
repack_filebuf_url	varchar(2000)
repack_fseq	numeric(20)

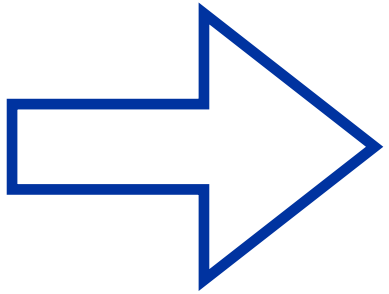
Intentional straightforward code development

- ensures **high performance IF** DB features exploited smartly (e.g. do not ask to count rows, write pop/delete counters)
- **requires optimisation** efforts per use-case
  - relies on the dev diligence with DB queries and DB admin tuning

CTA Scheduling Operations last year

ObjectStore Experience

- **fire-fighting**
  - 5 high priority dev tickets created in the last year
    - object **deletion** #309
    - empty shard **handling** #500
    - infinite **loops** #602
    - **locking** issues #460
    - **repack** exhausting OStore **resources** #573



Relational DB

- **challenges**
  - non-FIFO **priority queues**
  - object structure **("schema") updates**
  - CTA Scheduler code logic **not easy to extend/modify**
    tailored to ObjectStore backend structure (handling object dependencies)

CTA Request Ingestion

Each *taped* looks at all jobs for all drives
to get all (existing/hypothetical) Mounts
(+ 1st w/o global lock (DryRun) 2nd time with)

ObjectStore

- **summary objects** including regularly updated counters
- **locking** + multi-threaded access
- EOS MGM → cta-frontend **ingestion** one by one

Relational DB

- table views and counters
  (counters to be implemented if needed, we can avoid counting rows in queries)
- MVCC, explicit table/row locks, advisory locks
  (more about this later ...)

  smart locking might save us the DryRun
- **idea of bulk inserts if needed**
  (hold set of WFE requests until all in DB)

Tape Drive Efficiency and Data Integrity

Tape Free Space

- vendor tape **raw capacity understated**
  - by 1-5%, ~450 GB (?), *stable over time or decreasing ?*
- tape drive writes until **hitting tape end** !
  - **flush** tape writes in **bunches** of 200 files / ~32 GB (hard-coded)
  - last incomplete batch → failure; *time spent writing today **?***
- **cost-effective** (tape is cheap and drives fast today)
  - "waste" max space and time writing 32 GB per tape << extra free space

Tape Head Position Check

- there is a SCSI command to query tape drive position
  - avoids unnecessary flushes (Eric's idea)
  - IBM's approval needed to confirm read head position is indicative of what the write head wrote !

Fine-tuning not worth the effort today !

summer student study ?

home.cern