The future of FTS

XRootD and FTS Workshop 2024 at STFC UK

Steven Murray on behalf of the CERN FTS team

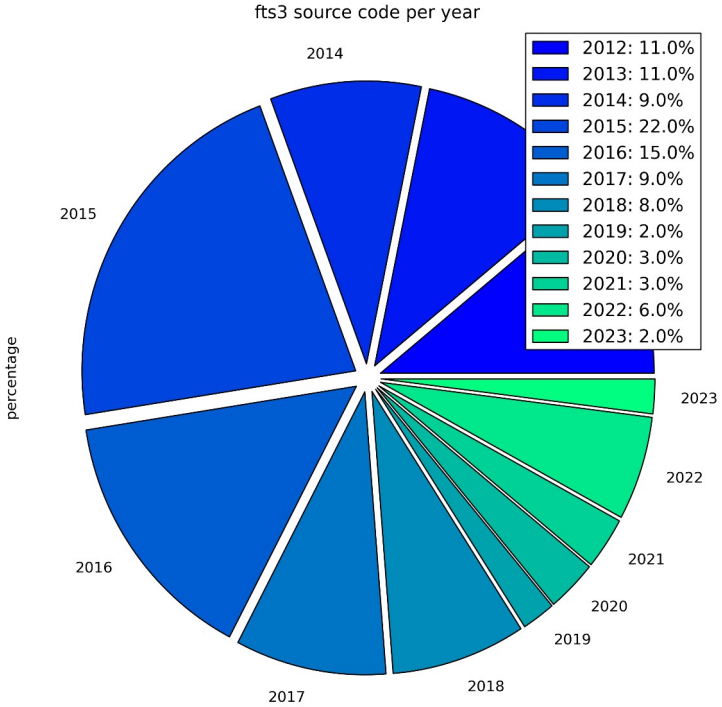
September 2024

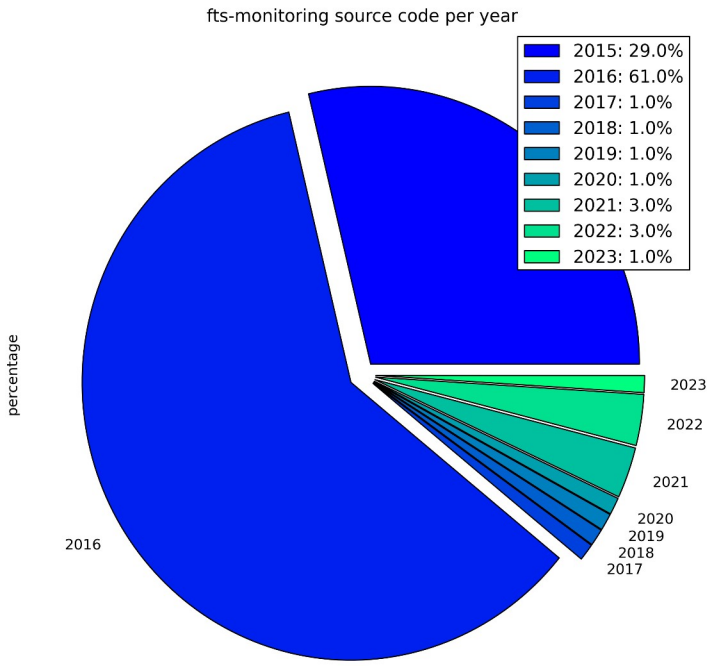
What’s wrong with FTS?

FTS
File Transfer Service

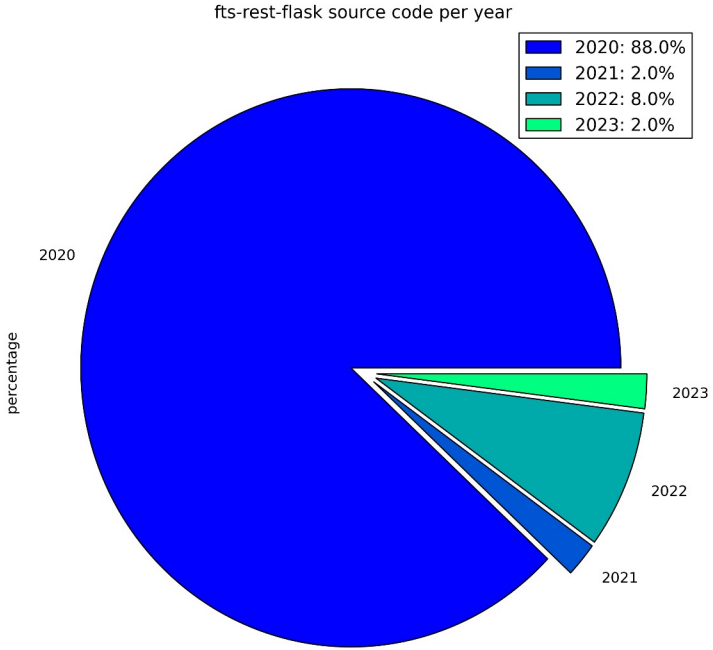
- **Legacy code base**
- **Too much C++**
- **We use MySQL when our collective expertise is in PostgreSQL**
- **Scheduler and optimizer overload the DB**
- **Many scheduling problems**
- **Missing features**

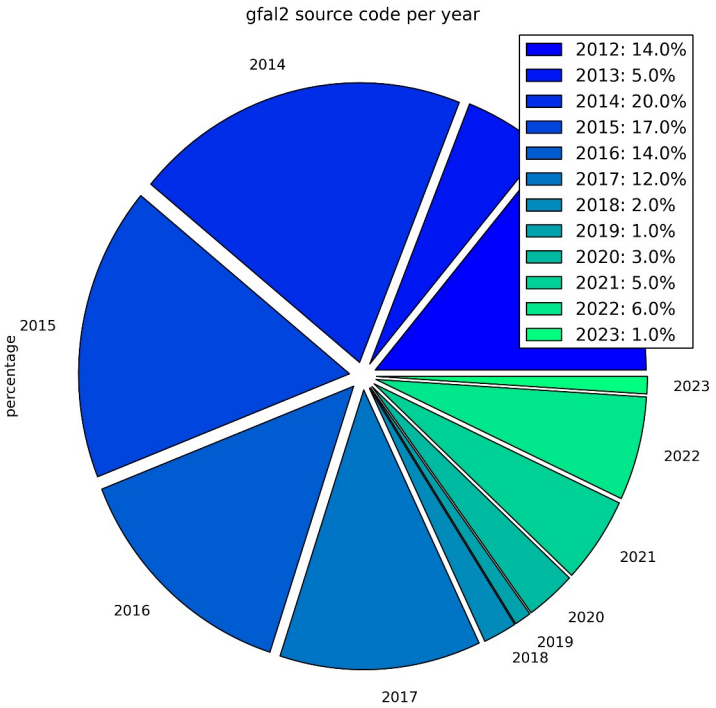
Legacy code base 1 of 3



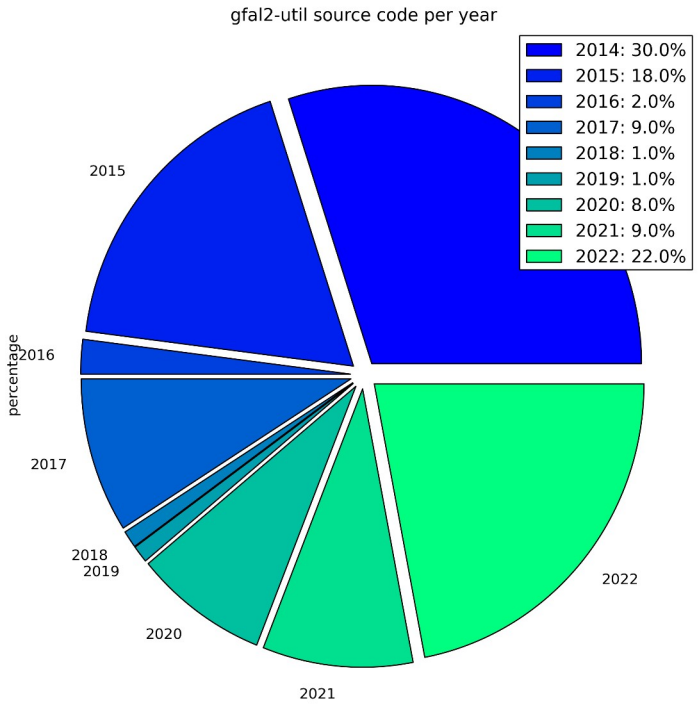


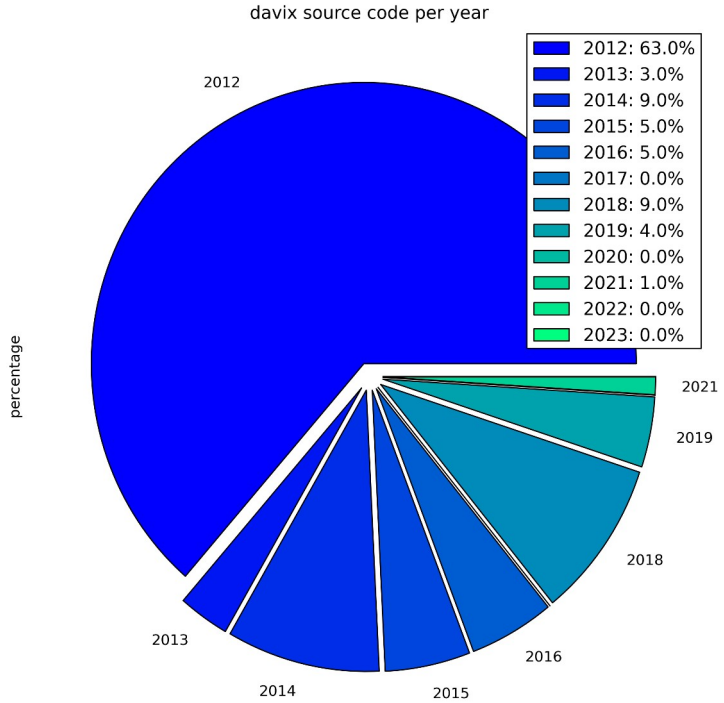
Legacy code base 2 of 3





Legacy code base 3 of 3





Too much C++ - Moving towards Python

- **Easier to hire staff with Python skills**
- **Easier to prototype new ideas in Python**
- **The following FTS daemons are written 100% in C++**
  - **`fts_msg_bulk`** – sends monitoring messages to ActiveMQ
  - **`fts_qos`** – manages tape transfers
  - **`fts_server`** – manages disk transfers
- **Everything can be rewritten in Python except for the file-transfer code**

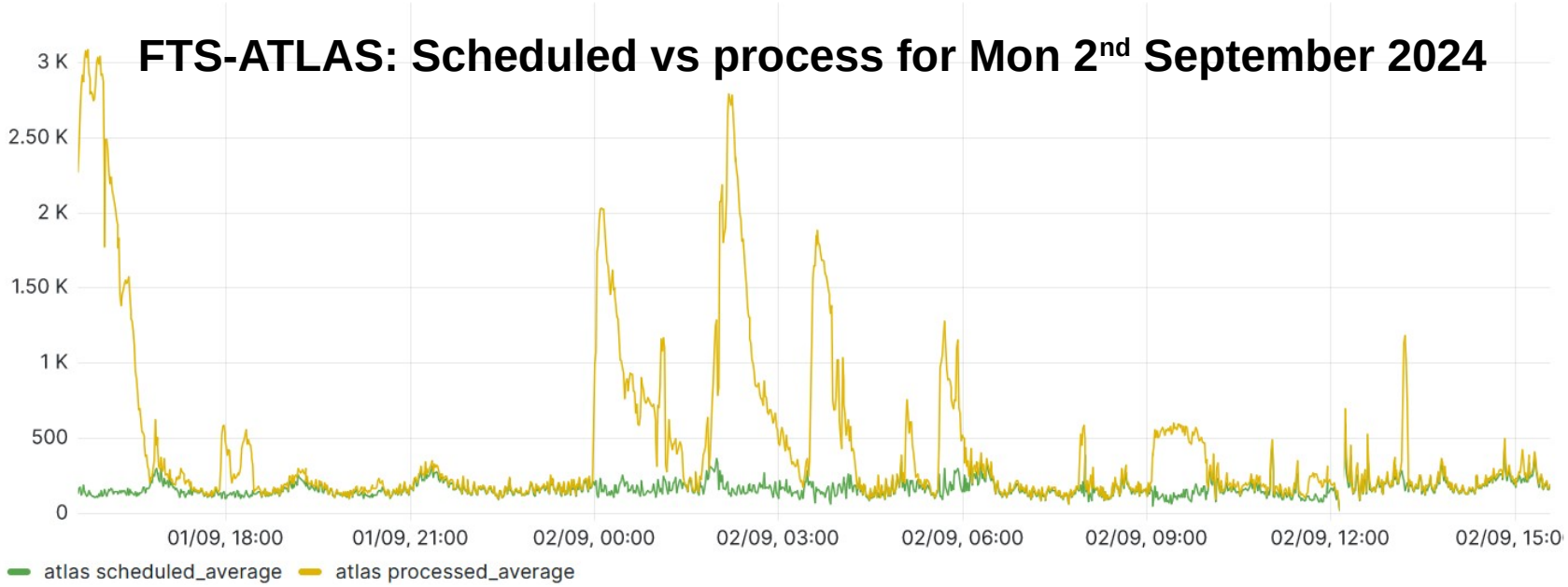
Moving towards PostgreSQL

FTS
File Transfer Service

- Cold start performance problems with MySQL on top of NFS – unavoidable DBoD setup
- PostgreSQL is seen by the European Commission as a credible alternative to MySQL
  - https://ec.europa.eu/commission/presscorner/detail/en/IP_10_40
- No single entity behind PostgreSQL
- PostgreSQL provides more index types allowing for more scalability options
- Multi-language support for stored procedures
- The IT Storage group is responsible for the CERN Tape Archive (CTA) project
- CTA uses PostgreSQL (and Oracle)
- The IT storage group have close ties with the ALICE experiment
- ALICE have a wealth of experience with PostgreSQL

The FTS scheduler needs to be replaced

- **Main reason – Amnesiac scheduler**
- **Scheduler immediately throws away all of its scheduling results**



Amnesiac scheduler overloads DB

- **Good: FTS uses a replica DB to improve its performance:**
  - Main DB for queuing, configuring and scheduling
  - Read-only replica DB for monitoring
- **Bad: Main DB load is still too high:**
  - Scheduler retrieves many possible transfers, executes a few and forgets the rest
  - Scheduler stats are always recalculated using hundreds of thousands of rows:

```
sql << "SELECT MAX(priority) "
   "FROM t_file "
   "WHERE "
   "     vo_name=:voName AND source_se=:source AND dest_se=:dest AND "
   "     file_state = 'SUBMITTED' AND "
   "     hashed_id BETWEEN :hStart AND :hEnd"
```

Many scheduling problems

1) Link vs link starvation
2) Throughput starvation
3) Cannot specify source shares when writing to a common destination
4) Cannot prioritise geographically closer links of multi-source transfers
5) No concepts for tape archive and retrieve staging areas
6) Fake distributed index-partitioning – many FTS schedulers with no shared global-view :
   - Transfer limits are exceeded
   - Non-FIFO ordering
   - Unwanted randomness and debugging noise
7) Aggressive scheduling when reaching the upper limit of a storage endpoint
8) Recalled tape-files garbage collected before being used

Link vs link starvation - implicit ordering in C++ data structures

- **When few transfers jobs can be started, implicit lexicographical order picks the same links(s) each time the scheduler is executed:**

```
boost::optional<TransferFile> TransferFileHandler::get(std::string vo)
{
    // get the index of the next File in turn for the VO
    boost::optional<FileIndex> index = getIndex(vo);
```

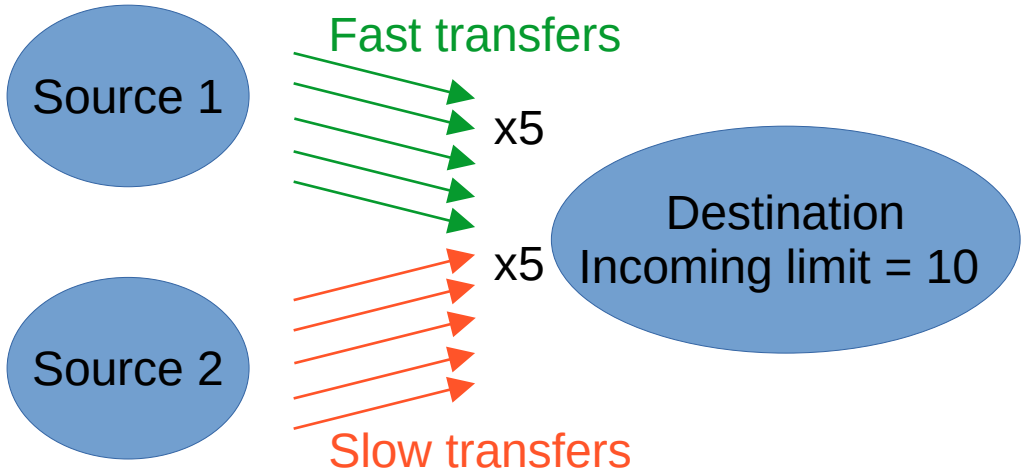
The map keys are lexicographically ordered source and destination pairs

```
boost::optional<FileIndex> TransferFileHandler::getIndex(std::string vo)
{
    ...

    // find the item
    std::map<std::string, std::map< std::pair<std::string, std::string>, std::list<FileIndex> > >::iterator it =
        voToFileIndexes.find(vo);
    ...
    // get the index value
    FileIndex index = it->second[*src_dst].front();
    it->second[*src_dst].pop_front();
```

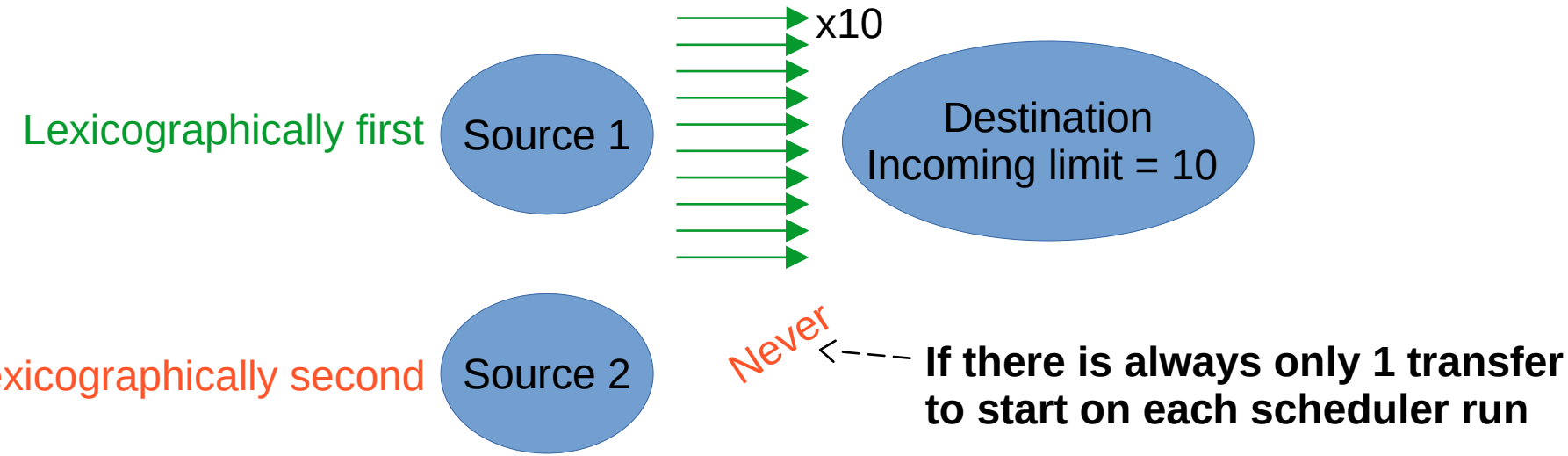
Throughput starvation - slow and fast transfers treated as equals

- **Multiple sources writing to a common destination**
- **Destination is NOT saturated**
- **Equal number of concurrent transfers given to each source**
- **Throughput of each link is ignored**



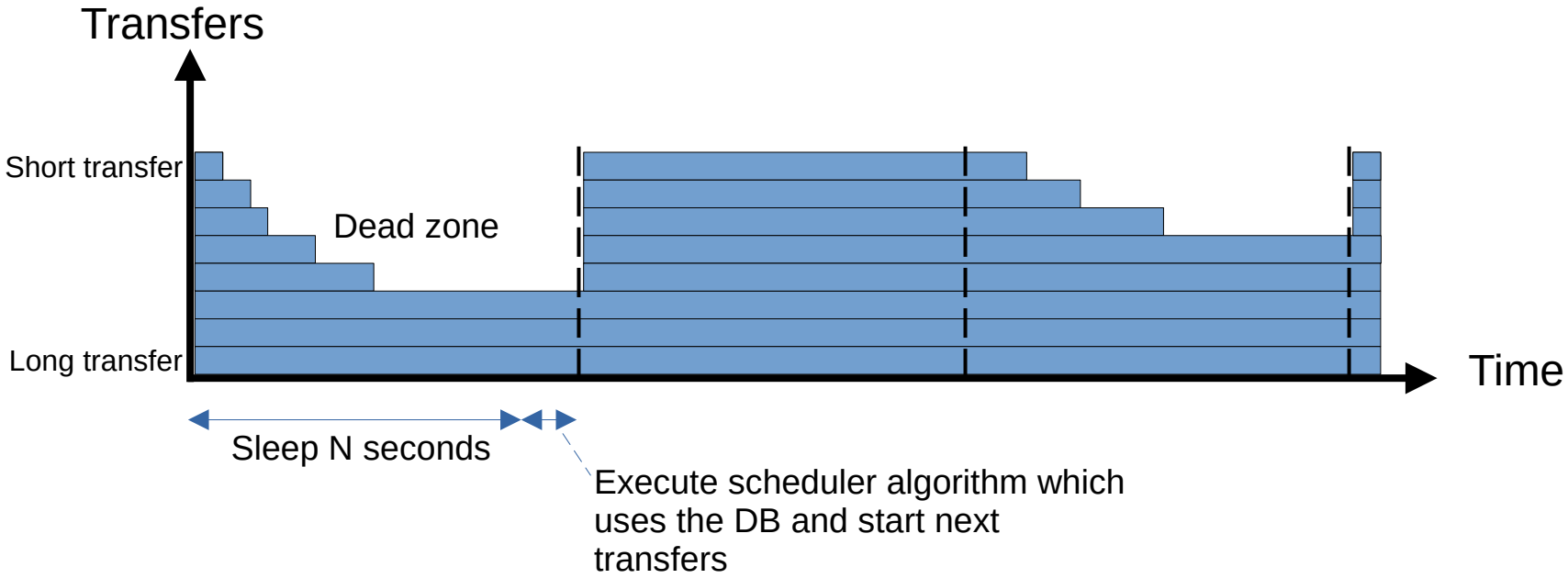
Throughput starvation – lexicographical starvation

- **Multiple sources writing to a common destination**
- **Destination IS saturated**
- **Very few transfers can takes place**
- **Some sources are starved due to lexicographical order of source/destination pairs**



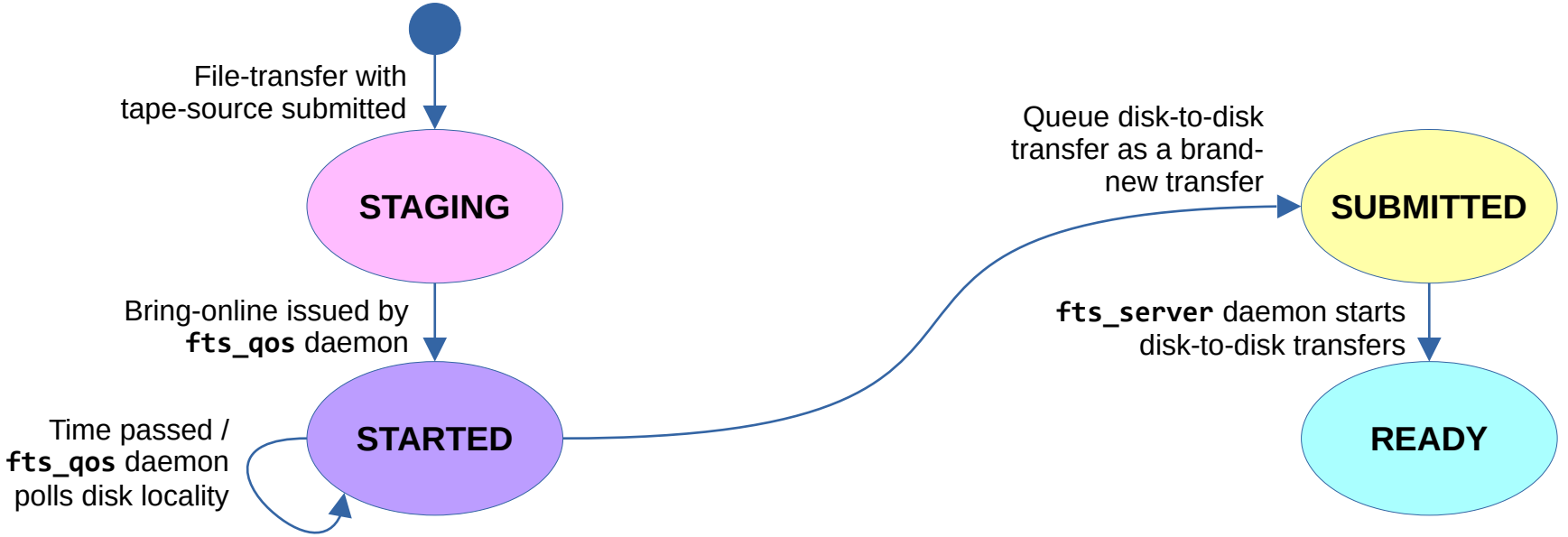
Throughput starvation – transfers started at fixed intervals

- **A transfer should be started when a previous one finishes**
- **Slow DB increases the “dead zone”**



Recalled tape-files garbage collected before being used

- **File-transfers with a tape-source are split brain**
- **`fts_qos` and `fts_server` are unaware of each other**
- **Nothing prevents `fts_qos` from retrieving a large amount of files from tape**
- **`fts_server` may or may not be able to read them out before they are garbage collected**



Possible additional features for the future

FTS
File Transfer Service

1) Support atomic uploads where supported by storage endpoints

2) Support for copying directory trees

3) Scheduling across multiple FTS instances – multi-instance constraints

4) Replayable transfers for debugging the Grid

5) Storage endpoint health monitoring

6) Million file user jobs – jobs are currently “small” batches for the FTS REST API

7) We hope to hear your feedback during this workshop

The plan – The steps to get to “FTS4”

1) Move from MySQL to PostgreSQL

2) Implement a dedicated FTS scheduler daemon in Python

3) Use DB in a scalable way

4) Add new functionalities

5) Migrate as much code as possible to Python

Current progress

- **Ported all MySQL queries to PostgreSQL**
- **An “empty” scheduler has been written in Python**
- **First scalable uses of DB have been implemented**

Scalable DB use

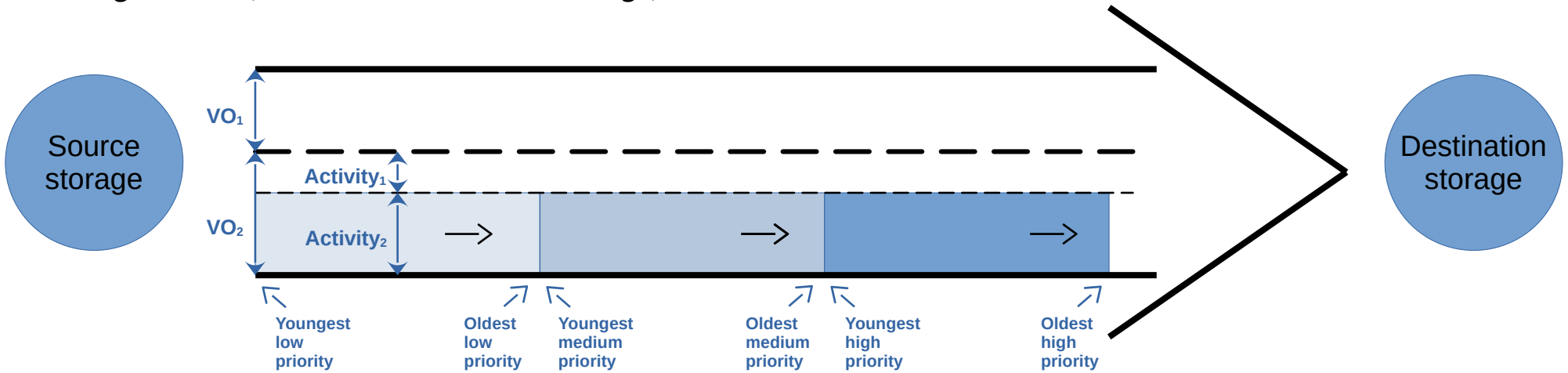
- **Reduce database-server RAM requirements to the minimum**
- **When getting the next file-transfer:**
  - **No** in-memory sorting of queue contents
  - **No** in-memory statistics gathering of queue contents
  - The “Next” file transfer **must** be a read from a persistent DB index
- **When scheduling**
  - Scheduler statistics **must not** be calculated from queue contents
  - All scheduler statics **must** be running statics and stored in the database as such

First scalable statistics

- **In a single transaction the `fts3web` application:**
  - Inserts a file transfer and increments the **`SUBMITTED`** count of the appropriate queue
- **In a single transaction the “empty” FTS scheduler:**
  - Decrements the **`SUBMITTED`** count of the current queue and increments the **`SCHEDULED`** count of the next queue
- **In a single transaction the `fts_server` daemon:**
  - Decrements the **`SCHEDULED`** count of the current queue and increments the **`READY`** count of the next queue
- **In a single transaction the `fts_server` daemon:**
  - Decrements the **`READY`** count of the current queue and increments the **`ACTIVE`** count of the next queue
- **In a single transaction the `fts_server` daemon:**
  - Decrements the **`ACTIVE`** count of the current queue and increments the **`FINISHED`** count of the next queue

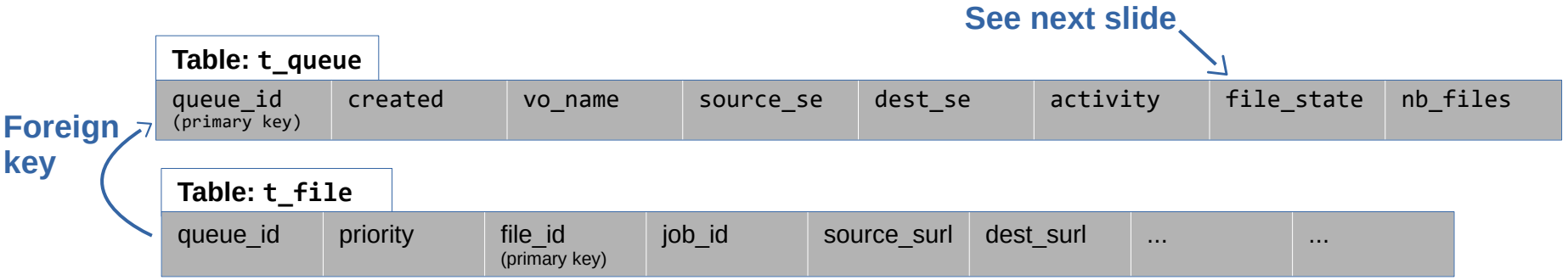
The anatomy of a queue

- **A queue is for:**
  - A given **activity** share for
  - A given Virtual Organisation (**VO**) over
  - A given **link** (source and destination storage)
- **Example use-cases**
  - Activity shares protect DAQ streams
  - Priorities expedite missing analysis files



The activity share vs priority debate is subjective. The above definitions are now frozen for FTS.

The new t_queue DB table



- The **idx_file_queue** index enables the DB to order queue contents on disk – no in-memory sort needed

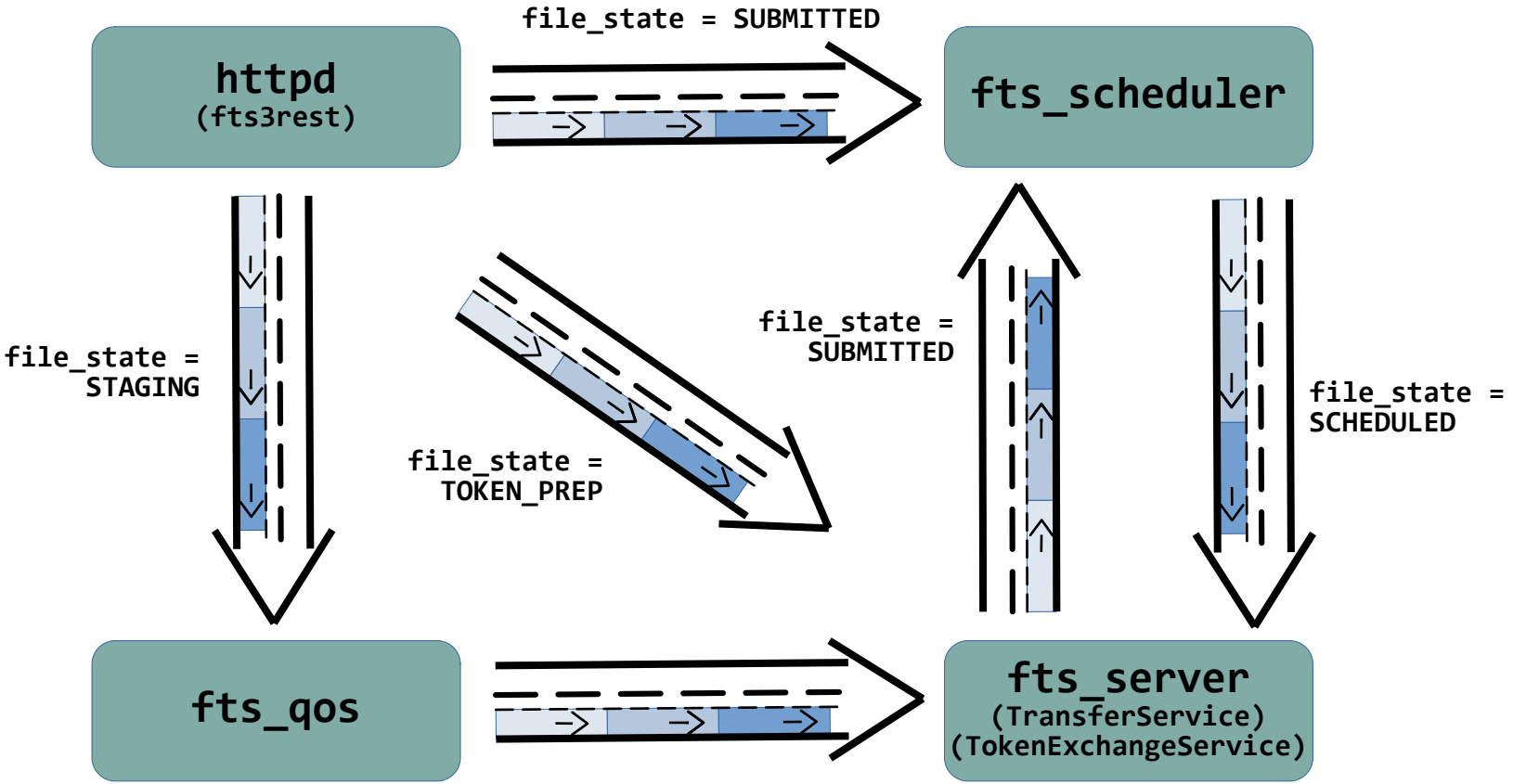
```
CREATE INDEX idx_file_queue ON t_file(queue_id, priority, file_id);
```

file_id = FIFO order

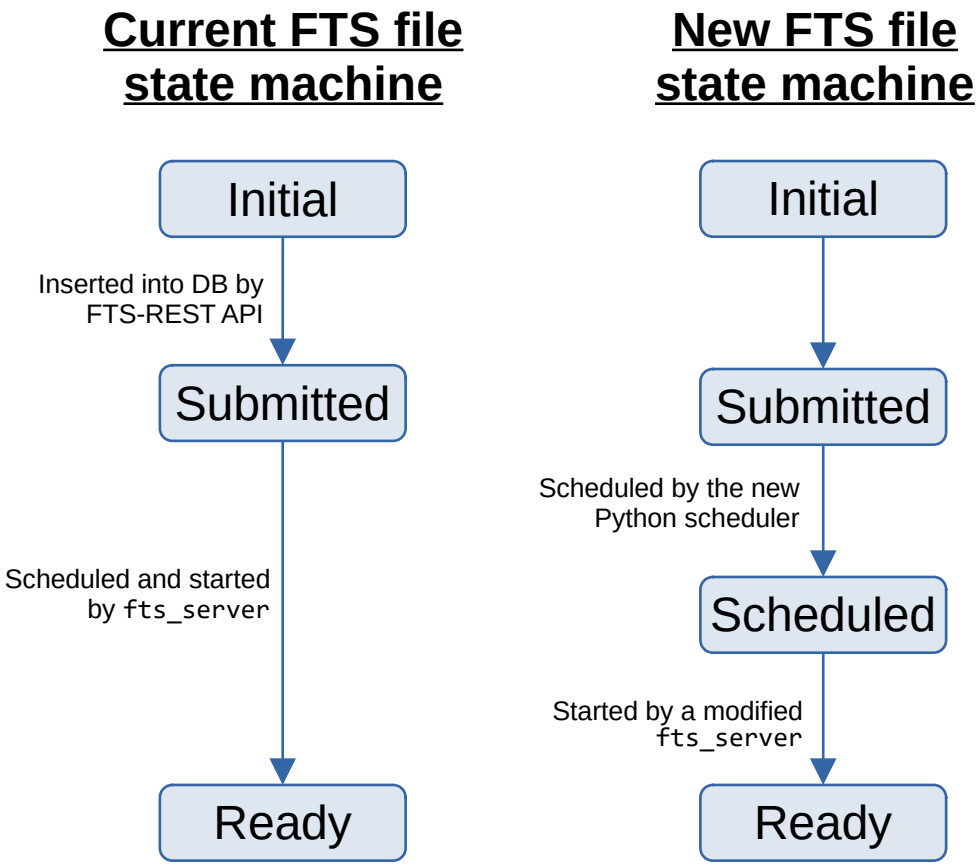
3 column index

- PostgreSQL recommends multicolumn indexes do not exceed three columns:
  - https://www.postgresql.org/docs/current/indexes-multicolumn.html
- **queue_id** allows the **t_file** table to not exceed a 3 column index
- The **t_queue** table allows as many identifying attributes as are necessary

Many sets of queues (file_state)



The new “empty” Python scheduler



The scheduler is currently “empty”

```
def schedule(dbconn):
    nb_scheduled = 0

    submitted_queues = get_queues_of_submitted_files(dbconn)

    for submitted_queue in submitted_queues:
        nb_scheduled += schedule_queue(submitted_queue)

    if nb_scheduled:
        log.info(f"Scheduled one or more transfers: nb_scheduled={nb_scheduled}")
```

Next steps

- Continue to implement the population of “running” stats in the database
- Close the scheduler feedback-loop – finished/failed transfer events
- Flesh out the new scheduler in Python
- Getting a minimum working solution in production at CERN ASAP
- Add new functionalities
- Convert as much C++ to Python as possible
- Make the scheduler algorithm pluggable?

home.cern