

Architecture of PFS

- Page file is a sequence of pages
 - A *page* is a basic unit of processing
 - All pages have the same length
 - A *page identifier* (abbr. `pgid`) points to the beginning of the given page (offset of a page in page file)
 - The page FS does not know anything about the contents of pages
- Architecture of Page file server
 - Page server is an Erlang process
 - * Page file server accepts requests from possibly many (ISAM) processes
 - * Process defines callback routines that define the messages (protocol of process)
 - * Process is connected by using data streams implemented in `query_node.erl` (maybe it should be renamed to `streams.erl`)
 - Binary file storage of pages
 - * N-th page is accessed by reading|writing from|to the position $n \times \text{page-size}$ in db file
 - * Pages are stored in binaries (unused fragments at the end of block)
 - * The question is weather the stream data pages are of the same size as file pages
 - * Read operation reads N pages from the given starting position in file
 - * Write operation writes N pages from the given starting position
 - * Append operation appends N pages to the end of data file
 - * Data is needed for read and write operation is transfered via data streams
 - * Protocol thus require the completion message for read and write (in opposite directions)
 - PFS is linked to a client vie I/O data streams
 - * Input/output data of write/read operations is obtained via data streams
 - * Data streams are composed of data messages that contain up to `TRIPLES_IN_PAGE` triples (or less)
 - * Reading/writing data messages/triples from/to a stream
 - Processing unit is either a data message or a triple
 - Stream type is defined on initialization of a named queue

- Requests are placed in a queue and served one by one
 - * Pid of the client process is stored for each request
 - * Request to read N pages is completed after all the pages are read and sent to client
 - * Request to write N pages starts after complete data has been transfered
 - Data can be stored in a map that maps Pids to lists of collected data pages
 - * Each request can process (read or write) a chunk of data
 - A chunk of data is defined by the number of pages
 - After a chunk is processed the state is stored in request and it is put back at the end of queue
 - This implements a kind of round-robin algorithm
 - All other request do not freeze if a large request is being processed
 - * (to-do) Does it make sense to have sessions (with a given process pid)?
- PFS protocol
 - * Protocol has only two states: **inactive** and **active**
 - After function **init()** is called on the creation of **gen_server**, the state is **inactive**
 - Message **start** moves the state to **active**
 - Read and write requests retain the state **active**
 - Message **stop** moves the state to **inactive**
 - * Synchronization between i) reading the new messages from a process queue and ii) executing the server requests
 - After a request, or a data processing slot of a request is finished the process queue is inspected
 - If there are no new messages the next request is processed
 - The control is given to the **gen_process** loop if there are no more requests
 - If there are new messages the process yields control to the system
 - Erlang runtime system picks the next new message and calls the callback routine
 - The callback routine first enters the request to the queue and then executes the next request
- (expand) A cache is part of PFS
 - * Page are read into buffer pool
 - * LRU page replacement strategy is used
- Page file server interface

– Incoming messages

* { data_read, Pid, PidData, Pgid, N }

- Pid is a pid of a client
- PidData is a pid of a process to receive data
- Reads a sequence of N pages starting at the page Pgid
- Read data pages are sent to the client process Pid via data streams

* { data_write, Pid, Pgid, N }

- Pid is a pid of a client
- Writes a sequence of N pages to the db file starting at the page Pgid
- Data pages to be written are received from a client process via data stream

* { data_append, Pid, N }

- Pid is a pid of a client
- Writes a sequence of N pages to the end of data file
- Data pages to be written are received from a client process via data stream

* { data_page, Pid, Data }

- Pid is a pid of a client
- PFS receives one data page Data from a client Pid

– Outgoing messages

* Pid ! { data_page, Pgid, Data }

- Server sends the contents Data of a data page Pgid to client Pid
- The receiver is set by the parameter Pid of data_read message

* Pid ! { data_read_end, N }

- Server signals to the client Pid the completion of data_read operation
- The data pages read have been sent to PidData
- The number of pages sent to client is N