

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
 - * PFS executes one client request at a given time
 - * Process implements callback routines that are triggered by 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*page-size` in db file
 - * Pages are stored in binaries (unused fragments at the end of block)
 - * The question is whether 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 needed for write operation is transferred via data streams
 - * Read operation requires sending the completion message to client
 - Client and the data destination may be different
 - See description of the operation `data_read`
 - PFS is linked to a client via I/O data streams
 - * Input/output data of write/read operations are obtained via data streams
 - * Data streams are composed of data messages that contain up to `3ples_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 transferred
 - Data can be stored in a map that maps pids to lists of collected data pages
 - When number of collected pages are **n** then writing starts
 - * 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 the request and the request is put back at the end of the queue
 - This implements a kind of round-robin algorithm
 - All other request do not freeze if a huge 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
 - If there are no more requests the control is given to the **gen_process** loop
 - 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
- A cache is part of PFS
 - * Pages are read into buffer pool

- Buffer pool is a map from `Pgid` to data pages read previously
 - * LRU page replacement strategy is used
 - A simple implementation using a double linked list
 - When a page is accessed it is moved to the beginning of the list
 - A given number of pages are maintained in the buffer pool
 - When the number of pages is more than the size of a buffer then pages are dropped from the end of the list
 - * Page access is changed when reading pages in PFS
 - Each page access checks first the buffer
 - No need to read pages from disk
 - All pages read from a disk are stored in the buffer
- Page file server interface
 - A client of PFS takes care of the construction as well as the decomposition of pages
 - * Pages provided as parameters of write operations have to be constructed outside PFS
 - * Pages received from read operation are decomposed outside PFS
 - * Functions for the construction/decomposition should be provided in PFS module?
 - Incoming messages
 - * { `data_read`, `Pid`, `Pgid`, `N`, `PidData` }
 - `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` }

- Messages `data_page` form are parts of a stream (`query_node.erl`)
 - `Pid` is a pid of a client
 - PFS receives one data page `Data` from a client `Pid`
- Outgoing messages
- * `Pid ! { data_page, Pgid, Data }`
 - Messages `data_page` form are parts of a stream (`query_node.erl`)
 - 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
 - For example, a client can be `isam` process and the receiver of data can be `tp_query_node` process
 - The data pages read are sent directly to `PidData` specified with `data_read` message (`tp_query_node`)
 - The number of pages sent to client is N