Tier 3 batch system data locality via managed caches

Max Fischer, Manuel Giffels, Christopher Jung, Eileen Kühn and Günter Quast

Karlsruhe Institute for Technology, Kaiserstraße 12, 76131 Karlsruhe

E-mail: {max.fischer, manuel.giffels, christopher.jung, eileen.kuehn, guenter.quast}@kit.edu

Abstract. Modern data processing increasingly relies on data locality for performance and scalability, whereas the common HEP approaches aim for uniform resource pools with minimal locality, recently even across site boundaries. To combine advantages of both, the High-Performance Data Analysis (HPDA) Tier 3 concept opportunistically establishes data locality via coordinated caches.

In accordance with HEP Tier 3 activities, the design incorporates two major assumptions: First, only a fraction of data is accessed regularly and thus the deciding factor for overall throughput. Second, data access may fallback to non-local, making permanent local data availability an inefficient resource usage strategy. Based on this, the HPDA design generically extends available storage hierarchies into the batch system. Using the batch system itself for scheduling file locality, an array of independent caches on the worker nodes is dynamically populated with high-profile data. Cache state information is exposed to the batch system both for managing caches and scheduling jobs. As a result, users directly work with a regular, adequately sized storage system. However, their automated batch processes are presented with local replications of data whenever possible.

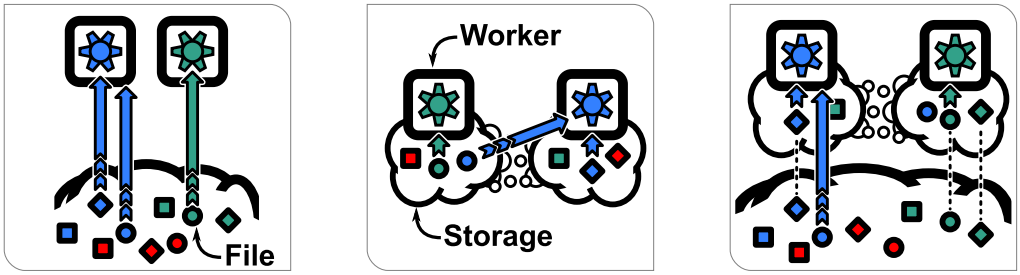
1. Introduction

The computing models of the LHC collaborations have been built around data storage [1, 2, 3, 4]. This is reflected in the hierarchical responsibilities in the LHC computing Grid, which defines the access and distribution of data. Processing of data in High Energy Physics (HEP) usually involves access from a dedicated computing cluster to a likewise dedicated storage cluster. Data locality, i.e. communication distance from data to processing, is typically provided at the granularity of an entire computing centre, housing both computing and storage resources.

In contrast, recent cluster computing architectures provide data locality at the granularity of individual computing nodes. Systems such as *GoogleFS* [5] or *Hadoop* [6] use mixed compute and storage resources: data is stored directly on compute nodes and processing is steered towards nodes hosting the appropriate data. By promoting local data access, data can be read at full capacity of the storage media while network communication is drastically reduced.

Due to the reduced network reliance, the mixed architecture generally scales better with increasing processing throughput: processing power can be added with little impact on existing resources. However, dedicated storage architectures are advantageous when the total data volume by far exceeds the momentarily relevant data volume: storage capacity can be increased without requiring an increase in computing capacity.

The 'High-Performance Data Analysis' (HPDA) design thus aims at combining the two paradigms. A top layer provides for fast local access, while a background layer serves as high-volume storage (see figure 1).



a) Dedicated Storage b) Integrated Storage c) HPDA Storage

Figure 1: Processing architecture illustration: Batch clusters in HEP usually use dedicated storage accessed via network from worker nodes (1a). Alternate setups such as *Hadoop* combine storage and worker nodes to allow local access (1b). HPDA provides local access to prominent data via caches overlaying classic high-volume dedicated storage (1c).

2. Conceptual Overview/Design Considerations

The workflows and data handling in HEP put several constraints on data usage. The HPDA design adopts two of these constraints as operation assumptions:

- Only a fraction of data is accessed on a regular basis with a high frequency.
- Non-local data access is efficient, provided it occurs with limited concurrency.

Therefore, data locality in the HPDA concept encompasses staging only the most relevant fraction of the total data locally on compute nodes. Any data can be fetched from remote storage, while prominent data is also kept available as a local copy; in essence, this is the functionality of a cache.

The deciding feature of HPDA caching is its scope: regular caching mechanisms for data from devices or hosts, such as Linux bcache [7] or CacheFS, operate individually on their single host. In the context of a pool of worker nodes however, all caches must likewise form a cache pool; state must be shared, e.g. to ensure even spread of data on all caches, to factor in imbalances of worker node capacities, or to decouple cache miss considerations from node availability.

In a cluster computing context such as HEP, data usage considerations extend beyond the current state of the system. For example, access to a portion of a data set implies an increased likelihood of subsequent access to the entire data set. Similarly, a history of repeated access to data suggests future usage beyond a classic cache expiration timespan dictated by data rate versus cache size (which would be low in the HEP context).

Thus, HPDA builds on the advantages of distributed caches [8], and extends it to actively coordinated caches. Caches are supervised globally, so data can be staged and released from caches asynchronously to current data accesses. The history of data accesses as well as features of external data are taken into consideration to formulate improved caching constraints.

3. Architecture and Implementation

While many individual features of the HPDA design have existing implementations, these are mostly part of specialized software or not compatible with HEP workflows. Hence, the HPDA core components are largely assembled as a custom implementation to ease flexibility and future

extensions. The batch system itself, storage access and cache mapping rely on software and frameworks common to the HEP community.

All custom components are written purely in Python due to the language's extensive standard library and the proven accessibility. *HTCondor* [9] was chosen as the batch system since it offers extensive means for communication between nodes; in addition, it is well established in CMS computing and under investigation by German CMS groups for opportunistic resources usage. For transparent access to caches and remote filesystems, the logical filesystems of *xRootD* and *unionFS* are used.

3.1. Exposed Caches

The HPDA cache on the worker nodes serves two functions: maintaining cached files and exposing their state to the batch system. Several elements keep the cache up-to-date and give it the autonomy to interact with the pool (see figure 2).

Meta-data of caches and files is stored in a *catalogue*. Without the need to explicitly query local or remote storage, the cache state can be easily inspected for allocation and exposed to the batch system. For failure robustness, regular checkpoints are used in combination with incremental updates. This allows to quickly reconstruct the current state of the catalogue from a checkpoint by replaying stored updates.

A *worker* thread constantly maintains the cached files by validating the meta-data against remote storage. If new files are added to the catalogue, the worker asynchronously fetches them for caching. Catalogue and worker communicate only with generic, simple file state and move instructions. Plugins for storage and cache APIs allow supporting a range of storage systems without adjustments to the core elements.

An additional *allocator* thread provides limited autonomy for the cache. The allocator scores files based on their features, e.g. access frequency. If there are several local caches, this is used for deducing in which to place the file. When a new file is suggested for caching, the score is used to determine if it is accepted and which older files to evict.

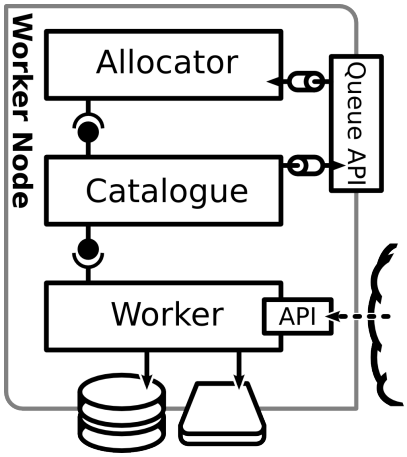


Figure 2: The HPDA cache consists of three main elements: The *catalogue* tracks the state of cached files. The *allocator* assigns files to cache media, creating fetching, movement and eviction tasks. The *worker* ensures validity of catalogued files and performs the allocator's tasks.

A generic API is used to exchange information with the batch system. The current content of the cache is regularly published to allow for data aware scheduling. Caching instructions are received via jobs and passed on to the allocator for validation; the job exit code is used to signal acceptance or rejection.

3.2. Coordinator

While the HPDA caches are able to share basic information, it is the responsibility of the coordinator to observe the big picture. It extracts resource requests from the job queue and history, and formulates staging orders to the cache pool if needed.

Information is extracted directly from the batch system, converting job ClassAds into attribute maps. Any feature of a job, such as expected I/O rate, can thus be considered for staging orders. The CMS grid overlay framework *glideinWMS* [10] has shown ClassAd extraction to be scalable to thousands of jobs even with a basic implementation.

Staging orders are dispatched in an anonymous fashion: An order encompasses the subject (i.e. file to stage), a score of importance, and resource constraints mimicking associated jobs. The assignment to distinct caches is the responsibility of the batch scheduler, while rebalancing and expiration is the responsibility of the caches. Thus, the coordinator does not need to be concerned with the topology of the worker and cache pool.

The optimal coordination strategy is of course a matter of optimisation. The current prototype implementation relies on manual staging orders; this is a logical side benefit of the need for manual management tools and sufficient for initial performance tests. Devising an optimal coordination strategy is the scope of future research.

3.3. Communication

In order to avoid duplicated infrastructure, HPDA components rely on the existing batch system for information exchange across hosts. With the *HTCondor* batch system, two forms of communication are used: ClassAds and communication jobs.

The ClassAd [11] is *HTCondor*'s mechanism of describing entities, such as worker nodes or jobs, with key-value attribute maps. In addition to standard entities, generic ClassAds with arbitrary attributes can be published globally in the batch system. Using this method, HPDA components can globally expose information about themselves, e.g. for reporting free cache space.

Since direct communication in the cache-coordinator pool is for caching requests only, it always encompasses scheduling problem: a fitting host must be found. To promote consistency with job scheduling as well as modularity and scalability, the HPDA components do not address each other directly. Instead, the information is encapsulated in a batch job with constraints on fitting recipients. The batch system then directs the information to a suitable host, if any. Once the job starts, it addresses the local HPDA component via Inter-Process Communication and reports success or failure of the request via its exit code.

For both methods, a thin wrapper API allows using them similar to message queues. This makes HPDA independent of the choice of the batch-system, as long as an equivalent API, possibly using external resources, can be provided.

4. Experience

While the HPDA design has not been fully implemented yet, experiences with individual components have been promising so far.

Access to local caches on mass storage easily matches network performance. Hard disk drives (HDD) are comparable to 1 Gb/s network, while a single solid-state drive (SSD) reaches more than half the throughput of a 10 Gb/s network. The largest issues are concurrent reads for HDDs, with degrading performance above four concurrent accesses, and the limited capacity of SSDs.

Interfacing with the *HTCondor* batch system has proven to be easy and reliable. Automated job submission and information extraction was extensively tested during the development of the *grid-control* [12] job management tool. The global *glideinWMS* infrastructure demonstrates the scalability of ClassAd information extraction far beyond the needs of HPDA.

5. Summary and Outlook

The High-Performance Data Analysis (HPDA) design aims at merging dedicated storage with data local batch processing. An array of caches on worker nodes forms a cache pool that provides local access to high profile data. The pool is coordinated globally to include external data and adjust to predicted access patterns. For scalability, a modular, decoupled architecture is chosen that adds only low overhead of coordination on top of regular batch system activity.

On the infrastructure level, future research will focus on the optimal data locality. With modern network, node-local and rack-local access is comparable in performance for some applications [13]. Motivated by this, parallel filesystems distributed on machines of the same racks are candidates for future improvements.

For high-level development, primary field of interest is the optimization of cache coordination. Especially recognition of access patterns to anticipate data usage for pre-staging is a promising approach.

Data locality versus high performance remote access is not a clear cut decision. Future scaling tests will thus investigate the optimal environment for either approach. Since the underlying *HTCondor* at KIT spans a non-flat topology of desktop computers, opportunistic resources, as well as closely and loosely storage attached workers, we expect to be able to compare a wide range of different dependencies.

Acknowledgments

The authors wish to thank all people and institutions involved in the project Large Scale Data Management and Analysis (LSDMA), as well as the German Helmholtz Association, and the Karlsruhe School of Elementary Particle and Astroparticle Physics (KSETA) for supporting and funding the work.

References

[1] Grandi C, Stickland D, Taylor L et al. 2004 The CMS computing model preprint CERN-LHCC-2004-035/G-083

[2] Jones R and Barberis D 2008 The ATLAS computing model *J. Phys.: Conf. Series* **119** 072020 doi:10.1088/1742-6596/119/7/072020

[3] Brook N 2004 LHCb computing model CERN-LHCb-2004-119

[4] Carminati F et al. 2004 ALICE computing model CERN-LHCC-2004-038/G-086

[5] Dean J and Ghemawat S 2004 MapReduce: Simplified Data Processing on Large Clusters *OSDI'04: Sixth Symposium on Operating System Design and Implementation*

[6] The Hadoop project homepage **URL** http://hadoop.apache.org/

[7] The bcache project homepage **URL** http://bcache.evilpiepirate.org/

[8] Paul S and Fei Z 2001 Distributed caching with centralized control *Computer Communications* **24-2** 256-68 (doi: 10.1016/S0140-3664(00)00322-4)

[9] Thain D, Tannenbaum T and Livny M 2005 Distributed computing in practice: the Condor experience *Concurrency Computat.: Pract. Exper.* **17** 323–56 (doi: 10.1002/cpe.938)

[10] Sfiligoi I, Bradley D C, Holzman B, Mhashilkar P, Padhi S and Wurthwein F 2009 The Pilot Way to Grid Resources Using glideinWMS *WRI World Congress on Computer Science and Information Engineering* **2** 428–32 (doi:10.1109/CSIE.2009.950)

[11] Raman R, Livny M C and Solomon M 1998 Matchmaking: Distributed Resource Management for High Throughput Computing *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing* 140–6 (doi:10.1109/HPDC.1998.709966)

[12] The Grid-Control project homepage URL https://ekptrac.physik.uni-karlsruhe.de/trac/grid-control/

[13] Ananthanarayanan G, Ghodsi A, Shenker S and Stoica I 2011 Disk-locality in datacenter computing considered irrelevant *Proceedings of the 13th USENIX conference on Hot topics in operating systems* 12 (doi:10.1109/HPDC.1998.709966)