A Scalable Object Store for Meteorological and Climate Data

Simon D. Smart
European Centre for Medium Range Weather Forecasts
Shinfield Park
Reading, United Kingdom RG2 9AX
simon.smart@ecmwf.int

Tiago Quintino
European Centre for Medium Range Weather Forecasts
Shinfield Park
Reading, United Kingdom RG2 9AX
tiago.quintino@ecmwf.int

Baudouin Raoult
European Centre for Medium Range Weather Forecasts
Shinfield Park
Reading, United Kingdom RG2 9AX
baudouin.raoult@ecmwf.int

ABSTRACT

Numerical Weather Prediction (NWP) and Climate simulations sit in the intersection between classically understood High Performance Computing (HPC) and the Big Data / High Performance Data Analytics (HPDA) communities. Driven by ever more ambitious scientific goals, both the size and number of output data elements generated as part of NWP operations has grown by several orders of magnitude, and will continue to grow into the future. The total amount of data is expected to grow exponentially with time, and over the last 30 years this increase has been approximately 40% per year. This poses significant scalability challenges for the data processing pipeline, and the movement of data through and between stages is one of the most significant factors in this. At ECMWF, meteorological data within the HPC facility is stored in an indexed data store for retrieval according to a well defined schema of meteorological metadata. This paper discusses the design and implementation of the next version (5th) of this indexed data store, which aims to increase the range of contexts within the operational workflow in which it can be used, and to increase its tolerance to failure. Further, it aims to pre-emptively head off some upcoming scalability bottlenecks present in the previous versions.

CCS CONCEPTS

• **Information systems → Indexed file organization**; *Hierarchical data models*; *Hierarchical storage management*; *Environment-specific retrieval*; B-trees; Version management;

ACM Reference format:
Simon D. Smart, Tiago Quintino, and Baudouin Raoult. 2017. A Scalable Object Store for Meteorological and Climate Data. In *Proceedings of PASC '17, Lugano, Switzerland, June 26-28, 2017,* 8 pages.
https://doi.org/10.1145/3093172.3093238

1 INTRODUCTION AND MOTIVATION

The European Centre for Medium-Range Weather Forecasts (ECMWF) is an international organisation whose principle objectives are the development and operational delivery of numerical weather forecasts, and the archival and distribution of meteorological data. Forecasts are produced in two one-hour time critical windows per day.

PASC '17, June 26-28, 2017, Lugano, Switzerland

ACM ISBN 978-1-4503-5062-4/17/06…$15.00
https://doi.org/10.1145/3093172.3093238

Meteorological output data is generated as a stack of 2-dimensional slices of the atmosphere surrounding the globe like layers of an onion. These slices are known as fields. For current operational work the fields each occupy between 1 and 20 MiB and they grow quadratically with resolution increases. In a one hour time-critical window the forecast system currently generates 20 TiB of output (7M fields).

The data is stored in byte streams described by GRIB format [23], which is standardised by the World Meteorological Organisation for compatibility between meteorological centres. This format is intricate, requiring specialised tools to decode and interpret, but self-describing, such that metadata can be extracted from a field. This metadata takes the form of a structured set of key-value pairs. Globally the potential metadata space is extremely sparsely occupied, but with very dense regions.

The Fields Database (FDB) is a software library, and an internally provided service, used as part of the weather forecasting software stack. In usage the FDB operates as a domain-specific object store for byte streams of meteorological data, such that the output from ECMWF's Integrated Forecasting System (IFS) is written into the FDB, from where it is retrieved by the various post-processing and archival tasks. In this capacity, the FDB also operates as the highest layer within an application controlled hierarchical storage manager. An overview of the overall computational infrastructure is given in Figure 1.

The Meteorological Archival and Retrieval System (MARS) [15] is a primary service offered by ECMWF that makes many decades of meteorological observations and forecasts available to wide range of end users. At the base of the stack is the tape archive built on the IBM High Performance Storage System (HPSS) [22] which is supported by the MARS disk-based cache. The FDB sits between the HPC systems and the rest of the MARS infrastructure, absorbing the forecast output and making it efficiently available throughout the forecast pipeline and elsewhere.

In practice, operational weather forecasts decay in value very rapidly after they are made (being superseded by forecasts made on later occasions). The FDB exists to make this data quickly and cheaply available while it is broadly useful and thus accessed frequently. Operational data in the FDB has a lifetime of between 3 and 5 days. Because the data flows through this system are predictable, they are application controlled and are thus optimised for performance.

In current operations, roughly 200 TiB of data pass through the FDB per day (including both operations and research activity). More than 100 TiB of this data is then moved to MARS for archival. At any given time, the total contents of the operational FDB are estimated between 4 to 5 PiB.

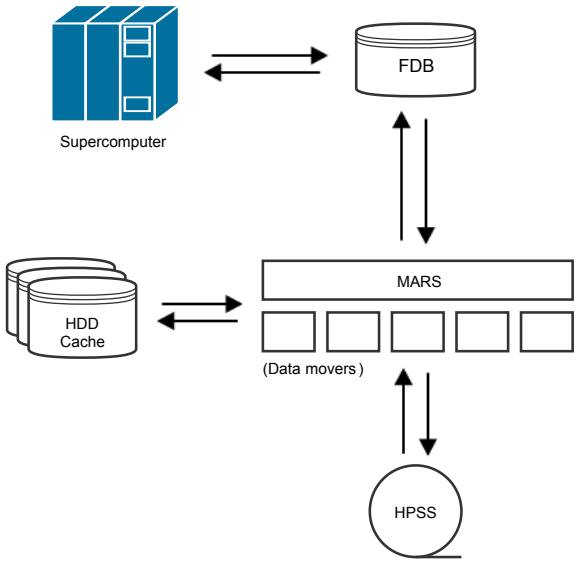

Figure 1: Overview of high-performance data infrastructure at ECMWF

The MARS infrastructure uses the key-value pair character of the metadata to inform its operational language. Data write requests use the complete metadata associated with a field to index it, and data accesses use either complete or partial sets of key-value pairs to navigate the indexing information and access the data objects. Which keys are required is well defined according to schemas that guide the data collocation policy.

There are a number of changes and challenges on the horizon. The amount of data generated and processed is not only large, it is growing rapidly. In 1995 operations generated a total of 14 TiB per year, and by the end of 2016 just the NWP model was generating 20 TiB in one hour. This is currently made up of 7M fields each of which are between 1 and 20 MiB. To support the scientific objectives and goals of ECMWF's 2025 strategy[6], both the resolution and diversity of the generated data must increase substantially. From historical trends, one third of the growth has resulted from increases in resolution, and two thirds from an increase in diversity.

I/O performance growth has not kept pace with the available computational capacity. Significant software engineering and changes to the use pattern of HPC hardware will be required to support the data volumes of the future. Further, the increase in diversity (and therefore number) of fields will increase the load on the metadata handling and indexing, thus making imperative that these scale.

A number of new technologies are likely to become available which change the available solution space. Commercial offerings are already moving high-speed SSDs into the cabinets of HPC systems to act as burst buffers[3], and the heterogeneity of the available offerings will only increase. ECMWF is particularly looking towards upcoming memory technologies including NVDIMMs.

In light of this, the FDB software has been rewritten as a more flexible component, with a particular eye to operating as a hot object store or an additional "hot" layer within the MARS infrastructure, whilst making use of the metadata and request handling capabilities of the MARS software.

1.1 Approaches to Data Storage and Curation

ECMWF considers management of data lifetimes to be key to its operational success. Systematically tagging and tracking the metadata associated with all generated meteorological data facilitates this process whether it is merely cached for seconds or archived for decades. The approach taken in this paper aims to marry up the performance and data curation considerations.

Many projects make use of MPI-IO[20] to facilitate high-performance I/O for large volumes of data on parallel filesystems. This is often embedded into the HDF5[19] library, and in turn into NetCDF[21] which provide a self-describing hierarchical container for data with attached metadata, which may follow such conventions as the Climate and Forecast (CF) metadata convention[5]. But this in itself does little to solve the data management and curation problem, and instead shifts this burden directly onto the application developers. As the WMO has specified the GRIB format in detail, with a well defined metadata schema, introducing HDF5 or NetCDF into the forecast pipeline increases rather than decreases the scale of our metadata handling challenges.

In the weather and climate field, the authors are not aware of other centres making use of indexed data stores within their data pipelines. Rather, the focus is on writing NetCDF files, or files containing GRIB messages, efficiently to known locations.

In contrast to typical HPC practice, where roughly three quarters of I/O usage is defensive in nature consisting of large blocks of checkpointing data, I/O usage in operational forecasts consists of very large numbers of independent fields. Accordingly, hardware parallelism can be exploited both by interleaving data from multiple nodes according to a defined pattern, by task-parallelising the data writing across a set of I/O server nodes, or a combination of both. One approach is demonstrated by the XML I/O Server (XIOS) developed for weather and climate data[12]. In an approach with multiple writers systematically writing non-conflicting data, the guarantees provided by a POSIX filesystem hinder performance more than they assist the application.

ECMWF is by no means alone in observing the difficulties posed by POSIX semantics to extreme-scale I/O involving many readers and writers. The most obvious direction to relax POSIX semantics is a move towards object storage, and this points in a direction of very active research at different scales. There are numerous systems providing object storage in different contexts[1, 2, 4, 7, 11, 16]. These ultimately reduce to various forms of hash table, with data being stored or returned according to a supplied key.

The focus of these general systems is performance and resilience on distributed commodity hardware, rather than the access and performance requirements of HPC. Further, ECMWF's data access requirements, particularly involving enumerated range requests, require a more extensive index which for these general purpose object store approaches would require an external indexing service.

Beyond these more general purpose object stores, more specific development is being invested into HPC solutions. The SAGE[18] project is working to explore the needs of extreme-data systems, and advanced object storage is being developed in conjunction with

Seagate[17]. The DAOS project[9] at Intel aims to comprehensively replace the I/O stack with a system built around object storage, with a great deal of flexibility for application developers to make use of novel data movement models.

Unfortunately these systems are far from being in use in operational centres. By contrast the system proposed in this paper offers object semantics on top of an existing parallel filesystem to enable a smooth transition of existing software running on existing hardware. Backensd built on alternative and upcoming technologies will be considered as they become available.

1.2 The FDB4

The FDB service has been in use for many years, and has undergone substantial evolution. It is heavily built into the workflow of a large suite of applications which work together to produce forecasts and climate simulations, containing millions of lines of legacy software. The FDB has proved to be robust and reliable in operations, supporting the data flows for 25 years, and this imposes significant robustness and resilience constraints on any replacement system. A new replacement system must be able to support existing data flows and interfaces whilst also looking to desirable upcoming changes.

The operational FDB is the storage layer for a multi-producer, multi-consumer system. A significant number of processes within the operational workflow are responsible for generating and indexing output data. Some of these processes are serial whilst others are parallel MPI simulations. Retrieval is performed by a distinct set of post-processing tasks running concurrently on different nodes. Each of these writers produces many fields, which are "close" to each other in terms of metadata.

GRIB data needs to be indexed for efficient access. Reading the metadata contained in each GRIB file involves a significant amount of decompression and decoding, and there are a very large number of potential fields. The FDB4 was built using an indexing scheme such that within each database (a subdivision of the overall metadata space) the index is a simple flat list of the written fields. As a field is written, its entry is appended or updated to this scheme list. In parallel runs, MPI is used to synchronise write access to this file. As the data being stored is dense and large, the ratio of indexing data to scientific data is very small (see metadata ratios in section 3).

Due to the use of MPI to synchronise access to the indexing data, only one MPI job can be writing to a given subsection of the FDB at any time. Although this restriction can be worked around by subdividing the data according to the distribution of data amongst the generating jobs, this places severe constraints on the design of the operational workflow, makes migration to different designs in operations difficult, and places the onus on the forecast system to ensure that multiple different experiments do not "clash".

The FDB4 implementation only supports rewriting with equal-sized data. In particular, because new data is written into the same location as the old data, if it has become larger it will overwrite any fields that come after it, resulting in data corruption. Unfortunately many simulations parameters affect the data size, such as resolution increase, different compression, increased precision, etc. This limits the scope of recovery actions that can be taken for a model rerun. Any dataset that contains data that might be modified must be entirely wiped and recalculated to ensure validity.

Finally, the FDB4 is non-transactional. If the any of the parallel processes fails during a simulation, it can result in corruption of the indexing data, resulting in the need to (at least) re-index the appropriate data and more likely rerun the entire set of calculations that write to that database. In practice this is currently a relatively minor problem in operations, as crash conditions are likely to result in a rerunning of the appropriate calculations anyway. However this does impose a severe restriction to the research workflow, forcing researchers to always reset an FBD database contents within the trial-error loop of running an experiment. To improve this, a proper transactional behaviour is necessary. Moreover, transactions would relax the current constraint that only parallel MPI processes may write concurrently into a single FDB database at any one time.

From an implementation perspective, although the FDB4 is working as a caching layer within the MARS ecosystem, it was written as an entirely standalone C library. As a result it does not share any of the rather more robust metadata, key, request and other library functionality contained in MARS. If the use of the FDB as a layer within the MARS ecosystem is to be extended with additional use cases, then it simplifies things significantly if the components share infrastructure code. For example, in the current scheme archiving data from the FDB to MARS involves a job that reads data from the FDB and then writes it to MARS, resulting in unnecessary intermediate files. The new implementation is able to make use of the MARS data moving protocol and can be instructed to move the data directly eliminating any intermediate data transfers, thus minimising I/O on the whole HPC facility.

2 DESIGN AND IMPLEMENTATION OF FDB5

The FDB5 aims to be an integral part of the MARS infrastructure. It needs to scale efficiently to arbitrary numbers of writer processes whilst retaining a continuously consistent state from the perspective of reading processes, irrespective of any potential failures. The design of the FDB5 exploits knowledge of the particular write and read patterns in both the research and operational workflows at ECMWF. In particular, each of the writing processes generates many fields with strongly related metadata, and read attempts are only triggered once the simulation has signalled to an external workflow controller that certain checkpoints have been reached (such as after a given iteration is complete), and this facilitates the chosen indexing structure.

In FDB5, each of these writers writes its own data and indexing information without reference to other processes, with the data only added to the global namespace once the writing process finalises its execution and the data has been flushed. A single reference is added to the end of a global Table Of Contents, with this step being a rare event and the the only one that requires synchronisation. This can be achieved using only the guarantees of the append operation on a POSIX filesystem, obviating the need for MPI or any explicit synchronisation or locking mechanism. This also permits the writers to be truly independent of each other. Deferring writing the indexing structure to disk until finalisation of the index, ensures that subsequent readers are accessing a read-only index

structure that won't be again updated, making the efficient loading and navigation of the metadata in-memory a trivial task.

Data written into the the database is also immutable, and the database is write-once, read-many. If data is modified then the changes are tracked in the metadata, and the modified field is conceptually different to the original resulting in it being stored as a new rather than a modified entry. Once a field has been written, any further writes using identical metadata mask the old data such that the new entry will be retrieved in a normal request, but the old data still exists. In doing so, potential data corruption associated with part-written elements is avoided, and the chain of events leading to the current state can always be safely unwound.

To future-proof the design of the FDB to cope with upcoming changes in the scientific and technological landscape, an active choice has been made to clearly delineate a boundary between the metadata handling and control in the front-end and backends which perform the storage. To implement the immediately required functionality, only one backend is required (operating on a POSIX-compliant filesystem, currently Lustre). However, the system has been designed taking into account the possibilities of upcoming technology, such as NVRAM, and the potential requirements to run a distributed service.

Due to its place in the larger MARS ecosystem, FDB5 is substantially built from pre-existing components. In particular the GRIB data handling and all the metadata and request handling services are reused. Further, much of the data server functionality exists within the MARS infrastructure, along with protocols to interact with remote clients. FDB5 is now built as just one aspect of the MARS server software and no longer is restricted to operating as a library compiled into components of the operational workflow. It could also now be configured as part of a dedicated caching infrastructure, such as on nodes containing SSDs or NVDIMMs within a larger HPC infrastructure, or with a distributed indexing and storage layout.

The newly developed FDB5 is currently undergoing internal evaluation and testing at ECMWF prior to being rolled out in production.

2.1 Hierarchical Indexing

Each field is uniquely identified by, on average, 10 key-value pairs within the metadata, with a large range of possible values for each pair. As a result, the potential metadata space spanned by the index is extremely large. Within this multidimensional space there exist extremely dense occupied subregions with the majority of the space remaining sparsely occupied. The indexing schema has been designed taking this into account. The keys for individual fields are mapped onto a schema which describes the permitted key structures and the relationship between the components. The values that make up a key are split into three groups, with decreasing sparsity of the metadata space, describing

The database These describe a larger dataset at a high level, capturing in particular the temporal identification of when the data was produced, and the overarching stream of data (such as all data from the high resolution midday forecast). In operational usage, a rolling window of databases are available, and as such an FDB will only contain a few dozen databases, which are likely to remain open for significant periods of time. In research, a new database is created per experiment.

The index A number of indexes belong to each database. These identify groups of fields that are likely to densely fill the metadata space.

The field The data elements being stored. From a storage perspective these are simple byte streams. As meteorological data is stored in GRIB format, the fields are self describing, and this description can be tested against the index for validation purposes.

Precisely how the key components map into the storage layer depends on the specifics of the back end in use. As each database is self contained it may make use of a different backend. The currently supported backend stores data and indexing information in a POSIX compliant filesystem. Further to this an experimental backend has been developed to store data in persistent memory on NVDIMMs using the pmem.io library[10] as part of the Horizon 2020 project NextGenIO [13]. Work is ongoing to build a backend that federates multiple FDB servers over the network, each possibly with different backend, into a single unified service using Rendezvous hashing [14].

Functionality in the backends is implemented using a visitor pattern[8]. The front-end knows how to walk through the schema, and the backend through the index, such that the specified functionality (for reading, writing, listing, etc.) is called back from the appropriate location. This allows a clean separation of the logical functionality.

2.2 POSIX (Filesystem) Backend

The POSIX backend is written to make use of the provisions of a POSIX compliant filesystem to prevent conflicts and permit writing to occur without requiring synchronisation. In operational usage at ECMWF this system runs on a distributed parallel filesystem, currently Lustre. To support this as efficiently as possible the number of accesses required to the filesystem metadata is minimised, with particular focus on the requirements of file locking.

Once an index has been opened for writing, a writer tends to append many fields to it before terminating. If this were not the case then adding indexes as a layer of indirection would not reduce the locking contention issues.

2.2.1 Structure. The database corresponds to a directory whose path is uniquely determined by the describing partial key. Each database is fully self-contained such that all of its data may be removed from an FDB simply by deleting the directory, simplifying management in operations. A visual overview of the directory structure is given in Figure 2. The primary access point for the database is its Table Of Contents (TOC). This file contains a sequential list of references to index files.

At the moment of creation, a copy is made of the global schema such that changes to that schema only impact newly created databases, whilst existing databases continue to operate unchanged.

Each index file contains an axis object. This describes succinctly the maximum extent of the occupied metadata space in the index. For each component of the key the spanned range of values is stored,

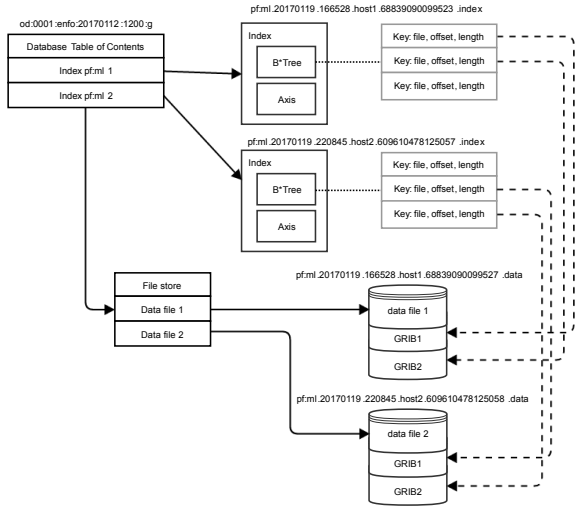

Figure 2: Conceptual layout of an FDB5 database

and when performing a lookup, a test against this axis object will exclude the majority of indexes from consideration.

Each index file contains a B*-tree mapping the key onto data references. These data references are composed of the specific file, offset and length where the data is stored. As GRIB messages are self-contained and self describing, these data files are merely concatenations of GRIB messages, and are thus valid GRIB files. As a consequence, it is straightforward to 'adopt' data in pre-existing files into the database without copying the data, or to re-index data if required. This also facilitates migration of operational systems from FDB4 as the data does not need to be moved or copied, but can simply be re-indexed and made available.

One index file is created for each writing process, and each of the data files belongs uniquely to a process. Due to a file naming scheme that generates unique filenames, no coordination is required between the writing processes during file creation.

2.2.2 Write Process. A write operation contains a (sequence of) GRIB messages and their associated metadata. For each field being written, the corresponding database is determined according to its key and to the current configuration. If this database does not exist it is created, otherwise the existing directory is used. An overview of the write process is given in Figure 3.

If the writing process has already created an index corresponding to the the key being written, that (open) index is used. Otherwise a new index object is created in memory, containing an axis object and a B*-tree as described above. Data files are used per writing process and per index. Once the appropriate index has been determined, the writing process selects the corresponding data file from its list of data files or creates a new one if it does not exist. The writing process appends the GRIB data and then adds the resultant location, offset and length to the index according to the supplied key. If the supplied key is outside the space described by the axis object, this object is updated accordingly.

The writing process keeps track of all indexes that it has touched. When a writer closes an index that it created (either at the end of program execution, or once a metadata-relevant point such as the end of an iteration has been reached) the index is flushed to disk. A reference to the recently finalised index is then appended to the database's TOC. The consistency guarantees built into a POSIX compliant filesystem will control the manner in which this data is appended and provide the required synchronisation or locking internally. This is the the only point of synchronisation between multiple concurrent writers. If the schema is well designed for the expected usage pattern the index finalisation events are sufficiently rare as to pose negligible load on the parallel filesystem's metadata handling, but this must be taken into account in configuration or pathological indexing performance may be observed.

From a consistency perspective, this means that data is not made available to reading processes until the writing processes have finalised their output. If a process fails between writing out data and finalising the index, then this data is not indexed, and potentially lost. Within the ECMWF workflow this is not problematic as the index finalisation coincides with the completion of model iterations which are the triggers for the various follow on processes that may read the data. This means, for example, that if the model suffers a failure while forecasting day 5, all previous forecasted days' data is guaranteed to be successfully indexed and persisted in storage.

2.2.3 Read Process. Read operations can be small or large. One of the characteristic features of the structure used for the keys, and the corresponding MARS requests, is that large numbers of fields can be requested in simple requests by including ranges or wildcards in the request.

A sample request is shown in Figure 5, containing both fully specified ranges (requesting the temperature field, and velocity components u and v), as well as automatic ranges (including model iterations from every 3 hours).

Informed by the current schema, the request is converted into a (series of) keys. The software is then responsible for enumerating only the occupied parts of the sparse metadata space, and ordering the data accesses optimally for the underlying hardware. An overview of the read process is given in Figure 4.

A visitor is passed to each of the databases which match the supplied request. These databases efficiently enumerate all matching entries within the sparse metadata space, and the visitor aggregates handles to the located data, which can then be (synchronously or asynchronously) fetched. Retrieving the data is deferred until all the index lookups have been carried out so that multiple accesses to the same data files can be reordered and streamlined for fastest access.

First the Table Of Contents (TOC) is mapped into memory. The entries in the table of contents are read in reverse order. When visiting the entries, the first entry with a given metadata matching the supplied key will be the one returned. As such, reading entries in reverse causes entries written later to 'mask' those written earlier, allowing for the correction of damaged or incorrect data without modifying or deleting it. Once a request has been fully satisfied, the exploration of the TOC and indexes can then be finalised.

Each entry in the TOC corresponds to an index. The axis object associated with each of these indexes is loaded, which describes the maximum extent in metadata space that it describes. If there is an intersection between the set of keys in the axis and the request

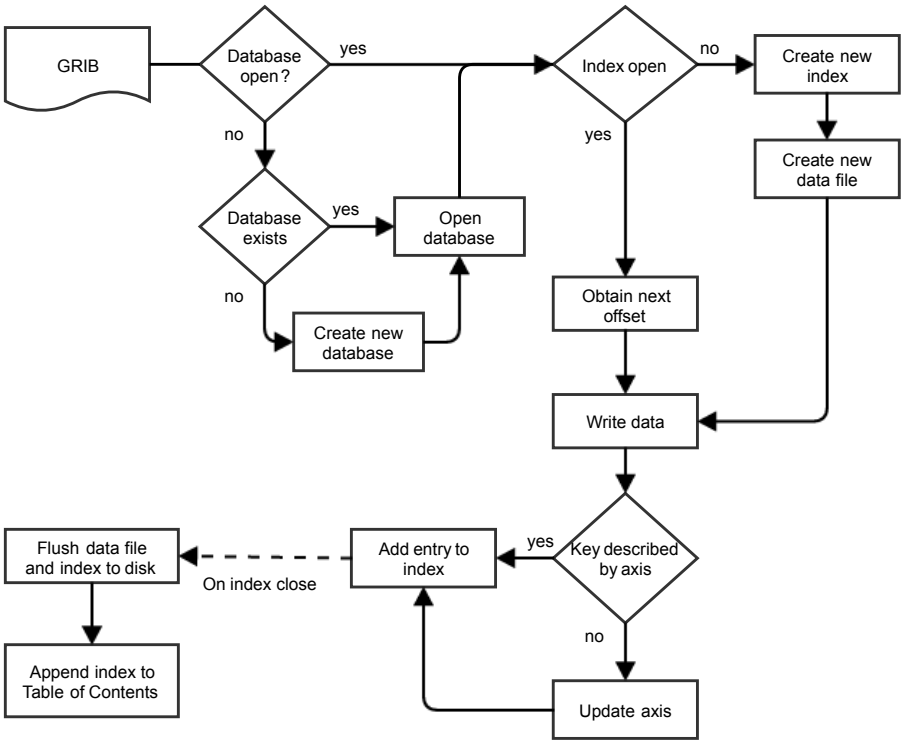

Figure 3: The write pathway

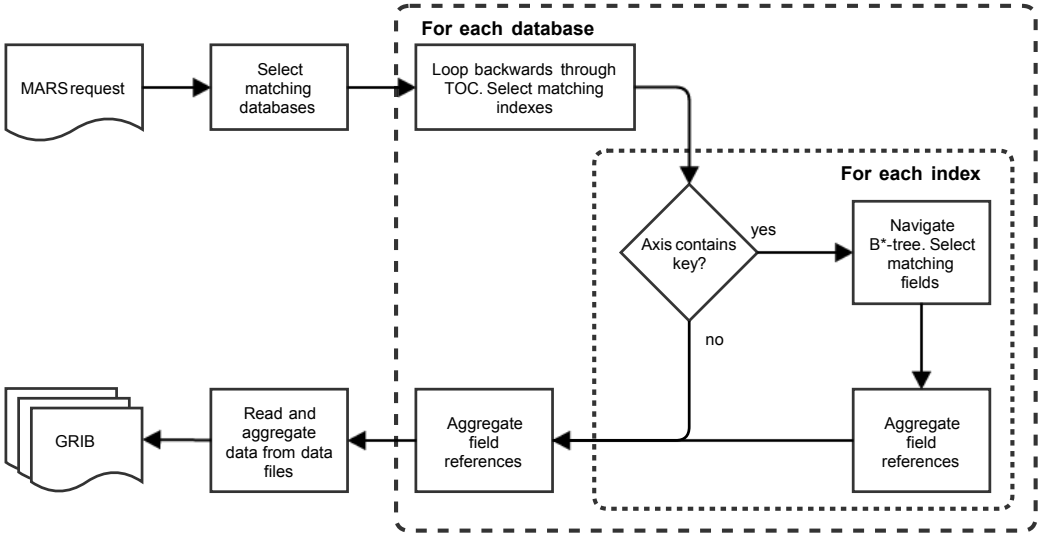

Figure 4: The read pathway

keys, then the index is opened, otherwise it is skipped. Inside the index the levels of the B*-tree are navigated to enumerate the fields matching the supplied request. The located field references are aggregated and returned as handles to the data.

Once the indexing information in all of the databases have been explored, the files referenced in the data handles are opened, data read from the specified offsets and concatenated into a a single byte stream containing the returned GRIB messages.

```
retrieve,
  class = od,
  type = pf,
  stream = enfo,
  expver = 0001,
  levelist = 1/to/90,
  param = t/u/v,
  date = 20170112,
  time = 1200,
  step = 1/to/240/by/3,
  number = 1,
  target = "output.grib"
```

Figure 5: Sample MARS request

Once databases and indices have been opened the most recently accessed ones remain cached in memory to accelerate following data accesses.

3 INITIAL RESULTS

Initial behavioural testing has been carried out using a 60 TiB dataset extracted from the operational workload. This data set contains the entire unprocessed output of the operational ensemble forecast for a single day.

Using the FDB4 implementation, the indexing information is made up of 879 index files, totalling 7.3 GiB (0.012% of the data size). When this data is adopted into an FDB5 database using the current schema it is split over 1954 index files totalling 3.7 Gib (0.006% of the data size). This large number of index files is due to the adopt process considering each subdivision of the FDB4 separately, and as such represents a worst-case indexing structure for the FDB5.

A subset of 82 GiB of this data (104k fields) was extracted and re-indexed into FDB5, resulting in a single index file (8.1 MiB, 0.009% of the data size). This is the most efficient indexing structure, both for reading and writing, that can be built using FDB5. This represents the best case scenario. As a result of this, we would anticipate that the performance of FDB5 in operations will likely be between these two limit cases. A further dataset was generated as the output of the first integration of FDB5 with the real output pathway of the ensemble forecasting system.

To give some flavour of the read performance of the systems, requests of various sizes have been carried out. There is a huge variability of performance depending on what data is in the file system caches (at multiple levels) and the background traffic levels on the system. To obtain the most reliably reproducibly numbers, the requests were carried out multiple times and the most rapidly completing outcome has been recorded. See Table 1 for read performance and Table 2 for write performance.

Reads for the adopted FDB5 were carried out using both the newly developed client tool, and using the wrapper tool which presents the same interface as for the FDB4. Both the FDB4 and the FDB5 acting on an adopted FDB4 database show (give or take caching effects) similar performance. When natively re-indexed the FDB5 provides significant read performance. It is clear that both the proliferation of indexes due to adopting an existing FDB4 cause noticeable performance degradation.

	Request time / s				
No. Fields	FDB4	FDB5 (FDB4*)	FDB5 (adopted)	FDB5 (reindexed)	FDB5 (written)
1	0.558	0.606	0.231	0.146	1.516
10	0.636	0.687	0.316	0.122	2.021
96	1.402	1.909	0.572	0.114	2.091
1080	9.384	15.482	3.855	0.639	6.514
10800	64.121	60.482	37.147	4.161	35.579
104400	3907.388	6472.621	5174.179	140.204	4339.211

Table 1: Comparison of read performance for different FDB systems.

* Access FDB5 (adopted) via FDB4 client interface.

System	No. fields	Time / s
FDB4	104400	410.46
FDB5	104400	379.768

Table 2: Comparison of write performance for different FDB systems

Note that because within this data set the different meteorological parameters have wildly different sizes, the data size does not grow particularly linearly with the number of fields requested.

These read and write datasets only compare single-process activity. In practice the usage of FDB5 will be in a parallel multi-writer context. The FDB4 and FDB5 have very different parallel behaviour, which is a partial motivation of the design of FDB5. As a result, the authors are aware that the timing comparisons are somewhat weak, and not entirely fair. What has been achieved so far is to validate that the underlying performance of the FDB5 is not significantly worse than the FDB4, and that the system works reliably. This will enable integration within a test version of the operational workflow, leading to the real side-by-side tests of two otherwise identical workflows using different online data stores. It is hoped that these tests will be completed in the upcoming months, and presented at the conference.

Considering the last column of the data, where data was generated using an actual forecast ensemble, the write performance was sensitive to the schema being used. It is essential to ensure that the rate of appends to the TOC does not exceed that which can be handled by the parallel filesystem, by controlling the number of writers appending to each database by splitting the data streams. If this maximum rate is not exceeded, then the IO impact on the forecast jobs is not noticeably different from when the FDB4 is used, being limited by disk throughput.

With a full dataset when written by a real forecast ensemble, the number of indexes generated that must be scanned to retrieve the data is somewhat larger than using the FDB4. This has two major impacts. firstly the performance is much more sensitive to filesystem caching. This is clearly observed in the request for 104400 fields, which overfills the cache resulting in poorer performance when repeatedly run than would be expected given the smaller

tests. Secondly, the absolute overhead of running read requests is higher, giving poor performance for very small numbers of fields. The authors intend to adjust the schema layouts to mitigate some of these issues.

4 CONCLUSIONS

The new FDB5 is currently undergoing acceptance testing at ECMWF prior to inclusion in the operational workflow. FDB5 brings a number of specific improvements over the operational FDB4.

Most importantly, the system has transactional write behaviour, such that failures occurring during writing processes will not result in a data store being left in an inconsistent state. This has been coupled with the ability for multiple, independent, writing processes to write to the same FDB without explicit synchronisation or coordination.

The system has been built with a strict decoupling between the front- and backends. This not only enables upcoming research into novel technologies and architectures, it permits experimentation with data layout and processing to facilitate different uses of the FDB (such as operating as an in-memory object store for data caching).

Using the POSIX backend, this system has performance characteristics that are designed for, and dependent on, the particulars of the data movements within the ECMWF workflow. Deferring writing the index files to disk, and adding their entries to the Table of Contents, until writer finalisation only makes sense in the context of a workflow where each writer outputs a large number of related fields close together in time.

As the FDB is relatively tightly coupled into the operational workflow (such that it is carefully optimised to work around the deficiencies of FDB4), it has been difficult to get meaningful throughput results in this paper. Making this worse, small test cases run essentially at hardware limited speeds with both systems, and meaningfully large test cases cannot be easily run without interfering with operational systems.

The true benefits of the FDB5 will be observed as the operational workflows are re-engineered to make use of options that simply did not exist previously. For example, the use of MARS software components, and in particular the data moving protocol, permits operational data, and more significantly research data to be moved out of the FDB into MARS without making use of further HPC resources.

ACKNOWLEDGMENTS

This work has been funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671951.

REFERENCES

[1] Amazon. 2016. AWS Storage Services Whitepaper. https://d0.awsstatic.com/whitepapers/Storage/AWS%20Storage%20Services%20Whitepaper-v9.pdf. (2016).

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267308.1267323

[3] Cray Inc. 2014. Cray XC40 DataWarp applications I/O accelerator. http://www.cray.com/sites/default/files/resources/CrayXC40-DataWarp.pdf. (2014).

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazonâĂŹs highly available key-value store. In *IN PROC. SOSP*. 205–220.

[5] Brian Eaton, Jonathan Gregory, Bob Drach, Karl Taylor, Steve Hankin, John Caron, Rich Signell, Phil Bentley, Greg Rappa, Heinke HÃűck, Alison Pamment, and Martin Juckes. 2011. NetCDF Climate and Forecast (CF) Metadata Conventions. http://cfconventions.org/cf-conventions/v1.6.0/cf-conventions.pdf. (2011).

[6] ECMWF. 2015. ECMWF Strategy 2016-2015, The strength of a common goal. http://www.ecmwf.int/sites/default/files/ECMWF_Strategy_2016-2025.pdf. (2015).

[7] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org/citation.cfm?id=1012889.1012894

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Intel. 2014. DAOS API Design Document. https://wiki.hpdd.intel.com/download/attachments/12127153/DAOS%202.5%20DAOS%20API%20and%20DAOS%20POSIX%20Design%20Document.pdf. (2014).

[10] Intel Corporation. 2017. pmem.io Persistent Memory Programming. http://pmem.io. (2017).

[11] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. DOI: http://dx.doi.org/10.1145/1773912.1773922

[12] Yann Meurdesoif, Ozdoba H., Caubel A., and Marti O. 2012. XIOS. http://forge.ipsl.jussieu.fr/ioserver/raw-attachment/wiki/WikiStart/XIOS_IO_Workshop_Hamburg.pdf. (2012).

[13] NextGenIO. 2016. NextGenIO website. http://www.nextgenio.eu/. (2016).

[14] University of Michigan. Department of Electrical Engineering, Computer Science. Computer Science, Engineering Division, D Thaler, and C Ravishankar. 1996. *A name-based mapping scheme for rendezvous*.

[15] B. Raoult. 2012. Architecture of the new MARS server. (2012).

[16] Redis MPI Forum. 2017. Redis Cluster Specification. https://redis.io/topics/cluster-spec. (2017).

[17] SAGE. 2016. Data Storage for Extreme Scale; The SAGE Project Technical White Paper. http://sagestorage.eu/sites/default/files/Sage%20White%20Paper%20v1.0.pdf. (2016).

[18] SAGE. 2016. Percipient StorAGe for Exascale. http://www.sagestorage.eu. (2016).

[19] The HDF Group. 2017. Parallel HDF5. https://support.hdfgroup.org/HDF5/PHDF5. (2017).

[20] The MPI Forum. 2012. MPI: A Message Passing Interface. http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. (2012).

[21] University Corporation for Atmospheric Research. 2016. NetCDF Format. www.unidata.ucar.edu/software/netcdf/docs. (2016).

[22] Richard W Watson and Robert A Coyne. 1995. The parallel I/O architecture of the high-performance storage system (HPSS). In *Mass Storage Systems, 1995.'Storage-At the Forefront of Information Infrastructures', Proceedings of the Fourteenth IEEE Symposium on*. IEEE, 27–44.

[23] World Meteorological Organization. 2013. GRIB Format. http://www.wmo.int/pages/prog/www/DPS/FM92-GRIB2-11-2003.pdf. (2013).