Layout-aware Exhaustive Search

Aravindan Raghuveer and David H.C. Du
DTC Intelligent Storage Consortium
University of Minnesota, Minneapolis, MN 55455

I. Introduction and Motivation

An exhaustive search operation, which examines all the files in a storage system, is required when an appropriate index is not readily available to quickly answer the query in hand. It is often extremely expensive, and sometimes infeasible, to construct indexes when the query model is fuzzy or when the data has high dimensionality. A standing example is content-based queries, like query by example (QBE), where the search is intended to retrieve the subset of objects that are *similar in content* to a given example. [2] presents a case for exhaustive content-based search on high-dimensional data like images and video. The authors point out that indexing and searching complex data is a challenging problem and the use of exhaustive search cannot be completely ruled out. So an exhaustive search operation is a “necessary-evil” feature that future filesystems and storage systems need to support.

We now present four key factors that spur us to design a new exhaustive search mechanism for modern disk drives:

1. Data Characterstics: Petabyte scale datasets are becoming increasingly common in scientific and HPC applications. Video surveillance can generate terabytes of data that need to be searched by content. In the personal computing paradigm, storage hungry applications like multimedia content creation and search are turning out to be killer apps.

2. Technology Trends: In recent years, disk capacity has improved rapidly while I/O bandwidth has not seen an equal increase. The effect of this capacity-bandwidth disparity on exhaustive search was succintly presented in [1]. According to the study in [1], it is predicted that with future hard disks, it would take at least a day to read 10TB of data sequentially and 5 months with random access (as 8KB chunks)!

3. Filesystem Design: An exhaustive search operation is not among the key design criteria of today’s filesystems. Hence a full filesystem search is implemented by recursively exploring the filesystem namespace (eg. grep). With filesystem aging and fragmentation, recursive exploration can end up being a random scan operation at the disk level. So on a full 10TB drive, a search operation will then take months to finish.

4. Concurrent Applications: The calculations in *factor-2* above make an implicit assumption that exhaustive search is the only application using the disk. However, in practice, a long running exhaustive search process will most likely be executing with other real-time filesystem applications. The impact of the exhaustive search process on the response time of other filesystem applications should be minimal.

II. Problem Statement

In this work, we aim to develop an exhaustive search algorithm that has two important properties as described below:

Layout-aware exhaustive scan: To achieve sequential access performance (eg. 24 hours to read 10TB) and to combat the ill effects of filesystem fragmentation, the search order of file extents should depend on their physical disk location rather than the filesystem hierarchy. Our search algorithm uses precise physical layout information to compute an optimal search plan to minimize overall disk head movement.

Suspend-and-Resume Search: A suspend resume model is required to service real-time requests wherein the exhaustive search is paused and resumed after servicing a real-time request. While resuming, the search plan may need to be modified taking into account the new position of the disk head.

III. Discussion

To achieve the two properties stated above, we are confronted with many questions and design tradeoffs. First of all, where should the exhaustive algorithm be embedded: the filesystem, an object based disk or a block device driver?

Next, in order to perform layout-aware exhaustive scan and suspend-and-resume search, two key pieces of metadata need to be maintained. First, *location metadata* about physical placement of data objects (eg. extent information) is needed to contruct the optimal scan schedule. The second piece of metadata, called *state metadata*, stores the progress of the exhaustive search operation. It consists of the list of objects that have been scanned so far. After a suspend-resume step, this list is used to avoid re-scanning objects that have already been scanned. What is the best way to represent the location and state metadata? Should they be stored as tables or should an object and its metadata be physically stored together?

The next question is how to place the metadata relative to the data on the disk. If all the metadata is placed in one central location on disk, we need to pay a seek penalty to read a batch of metadata before scanning the objects in the batch or when we need to dump the state metadata before servicing a real-time request. Alternatively, if we decide to store the metadata distributed over the disk, how should the metadata layout look like?

The final question that needs to be answered is about the search strategy itself. After servicing a real-time request, how should the exhaustive search plan be modified? Should we select that part of the disk that has not yet been explored at all or should we finish an area that is almost done?

References

[1] J. Gray. Keynote at FAST 2005. `http://www.usenix.org/events/fast05/tech/mp3/gray.mp3`.

[2] L. Huston et al. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of FAST*, 2004.