QUASAR: Interaction with File Systems Using a Query and Naming Language

Technical Report UCSC-SSRC-08-03
September 2008

Sasha Ames
sasha@cs.ucsc.edu

Carlos Maltzahn
carlosm@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

QUASAR: Interaction with File Systems Using a Query and Naming Language

Sasha Ames Carlos Maltzahn Ethan L. Miller
sasha,carlosm,elm@cs.ucsc.edu

Storage Systems Research Center
Computer Science Department
University of California, Santa Cruz

Abstract

As storage capacities increase, finding and organizing data becomes increasingly challenging. Conventional approaches to organization for file systems fail to effectively provide for the needs of petascale storage, because hierarchical namespaces do not scale and must rely on ad hoc utilities. Previous solutions do not incorporate relationships as search terms, nor were they designed for today's systems sizes. Moreover, adopting established query languages as file system interfaces is not practical because such languages are too complex for the common task of specifying file names, and file systems users would be forced to switch to a drastically different interface.

To address this problem, we present a query language (QUASAR) that combines a number of useful operations for search and view specification within file systems. QUASAR narrows down file system namespaces to smaller subsets, through searches based on link distance, and provides more meaningful results through searches based on inter-file relationships. Instead of being confronted with long lists of search results, users may employ QUASAR to define views for their searches, and then navigate views using the familiar abstraction of hierarchy browsing. In contrast to an abrupt transition to a different language, QUASAR provides a syntax that includes POSIX paths as well as added features that are suitable for naming. We demonstrate such features through a prototype file system that uses QUASAR queries to manage its namespace. We further discuss our thoughts on metadata typing.

1 Introduction

The growth in storage capacities has been accompanied by an equal explosion in the quantities of data stored. Such growth amounts to systems containing nearly one trillion files in the near future: a proliferation of metadata to manage. Quickly finding files in storage systems is always a critical need, and given their growth, this activity becomes increasingly challenging. For instance, large organizations often have personnel working independently on projects, and although the workers would benefit from sharing knowledge, communications are difficult under the current paradigm of file systems organization. Effective search techniques within common storage facilitate sharing. Moreover, the intelligence gathering community has enormous amounts of data (audio, video, photos and text) which need to be stored and retrieved. Finding the desired file quickly may be critical in a crisis. Relationships between such data items are valuable in aiding retrieval when preserved in the storage system [1, 2, 5].

Because the POSIX interface lacks a direct search capability, developers have addressed the issue of finding files through ad hoc search techniques or overlay applications. In this paper, we present an alternative approach, in which we place the interface for querying and specifying views of files within the kernel. We explore this approach with our query and naming language, "QUASAR", which combines established techniques for file system interaction with several new techniques as well as syntax for specifying views. QUASAR assumes a file system with rich metadata, including extended attributes on files, and relational links between files (see figure 1). Our previous work on the Linking File System [1, 2] explored this model further. Other prior work presented methods by which inter-file relationships of various categories might be extracted, including temporal locality [9], provenance [5] or causality [8].

2 Background

Users of the POSIX interface, which uses directory hierarchies for file organization, must rely on finding files by browsing. As hierarchical namespaces do not scale, such activity becomes impeded by an excessive number of traversals. While profiling a single laptop with 65GB

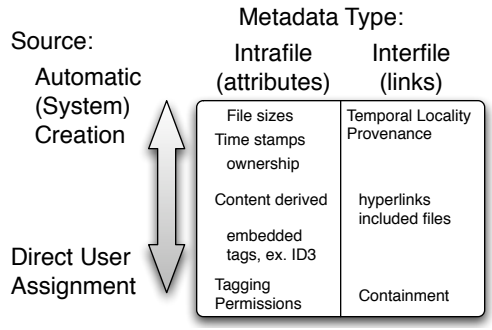

Figure 1: *QUASAR works with a rich variety of metadata. We categorize metadata on two axes, and view the variation of user vs. system involvement in its creation as a continuum.*

of data stored, we encountered a little under one million files, an average directory size of five entries, path lengths on average of ten traversals, and 29 maximum traversals. If we extrapolate five orders of magnitude to nearly 100 billion files, and consider a file distribution consistent with our directory size, browsing for files would require as many as sixteen traversals.

In response to the problem described above, semantic file systems presented attribute-based naming schemes. An early solution of this sort [3] allowed users to browse virtual directories of files tagged with attributes, and the logic file system [6] followed with logical operators within path expressions to produce virtual directories. Because such schemes use attributes associated with files, common attributes match many files, and rarer attributes only match a few files apiece. Performing search via a query using common terms produces desirable results only if the user specifies a good combination of terms, and guesswork is needed to compose a "good" query. Alternately, the use of rare attributes is worthwhile only if users can correctly remember these particular attributes: a failure to recall necessitates further browsing.

One way in which QUASAR approaches namespace scaling is through combining keyword search with hierarchical browsing. No other solution has successfully combined the two: file search results usually appear in a single virtual directory. In the solutions described above, virtual directories cannot contain within them additional virtual directories for the purpose of browsing. In contrast, QUASAR's views can contain hierarchies that users may further browse.

Within the UNIX community, many shell-based utilities have been developed for varying types of search, outside of the kernel. For instance, a variety of *search paths* are utilized, yet their values are not easily ported between environment variables. Moreover, different shells (bash, csh, etc.) often use different syntax, such as separator characters, for the same variable. In the case of LaTeX and *cvs*, if users wish to share BibTeX files across documents, they must repeatedly soft-link these files, stored in a separate cvs directory from their documents' workspaces. Another problem with shell-based solutions is that they have been invented on small systems, and thus are unable to scale to larger systems. For instance, many scientific applications write numerous small files to single directories. It is impractical to obtain listings of such directories, and brute force search techniques (find+grep) become prohibitively slow.

Most recently, we witnessed the proliferation of keyword search applications, such as Spotlight, Google Desktop and Beagle. These and the shell-based solutions have a disadvantage that, not being in the kernel, their performance may not be optimized within the file system. The search applications maintain their own indices and tie into the file system through coarse notification mechanisms or continuous crawling. However, as the kernel provides the mechanism to set the current working directory, hiding the complexity of long pathnames, QUASAR is able to provide (through the kernel) the same functionality as the utilities do, yet in a consistent fashion. We do so by changing the file system's naming interface. This approach guarantees a common medium containing richer file names, analogous to the VFS layer, which presently handles naming.

Another alternative for organizing file metadata is the use of relational databases, which we see as problematic. Their use ultimately requires adopting SQL as the file system interface, whose syntax appears too verbose to specify names for groups of files. Furthermore, if we must adopt the relational model for file systems in an extensible fashion, without a specific schema for *all* attribute fields, we must use a general schema that requires numerous joins to answer queries. In order to support existing applications' use of POSIX directories and our link distance search, we have determined that we would require extensive iterative querying, costly recursive joins, or even SQL union operations. Additionally, we have determined that other query languages, such as XQuery/XPath [4] and SPARQL [7] are unsuitable for adoption. XQuery is not appropriate, as it is solely for XML transforms. XPath is a node selection language, lacking view specifying operations that we require, and it also lacks facility for non-hierarchical data. SPARQL is a bad fit because, like SQL, it poorly handles traversals, and its syntax produces queries that will be far too long for naming, due to the nature of the RDF data model.

3 Flexible Naming and Virtual Hierarchies

We consider a number of classes of naming within file systems. First, the file name is the "listing name" that appears in directories. If a file is moved or copied to another directory, the file retains this name in its new location,

unless there is already a file present with the same listing name at that particular location. These names have a 1:N ratio (names:files). The second type of file name is the full path. A full path contains both the listing name and the traversal of all encompassing directories, with each step including a listing name for each corresponding directory. These names are N:1, where N is extremely close to 1, reflecting the very small number of hard linked files and directories. Finally, file systems internally keep track of files via their inode numbers, guaranteeing a 1:1 ratio. QUASAR makes a number of changes to the first and second described types of naming. For the first, QUASAR presents listing names for files, within both traditional and virtual subdirectories, comprised of one or more arbitrary attribute values. For the second, N becomes large, since paths to files include attribute search information in addition to traversals, and traversals may occur over more than one naming attribute. Also, inode numbers can be used programmatically within searches and result listings.

Directory hierarchies, when not too extensive, are generally effective for users to browse for desired information and applications for their own data storage needs. A major drawback of conventional hierarchies is that they remain static. We are forced to see the same view organization no matter what the circumstance. In contrast, search engines return their results in lists with little to no organization. This feature is a serious drawback to search engine scalability, because users become inundated with too many disorganized results. To address both these issues, we propose the use of dynamic hierarchies. We discuss their use in the following domain-specific example, illustrated in figure 2.

A number of audio file player and management applications have been developed to help users enjoy their digital music collections. Various formats, including .mp3 and .aac, embed song metadata within each file under the de-facto ID3 standard, but the applications still maintain their own queriable databases with additional non-standard metadata. These databases are problematic in that they are not portable between applications, and each application presents one or more interfaces for querying or for viewing results, each with inconsistent capabilities. For instance, iTunes has a main query and a smart playlist interface, which, respectively, have limited searchable fields and non-browseable results. Moreover, the naming schemes and the locations within directories are not consistent across applications with respect to usage of the underlying file system for storage, and of course, have the limitation of static hierarchical placement.

QUASAR overcomes issues with static hierarchy organization and limited view presentation by combining the positive features of each. Depending on how the user

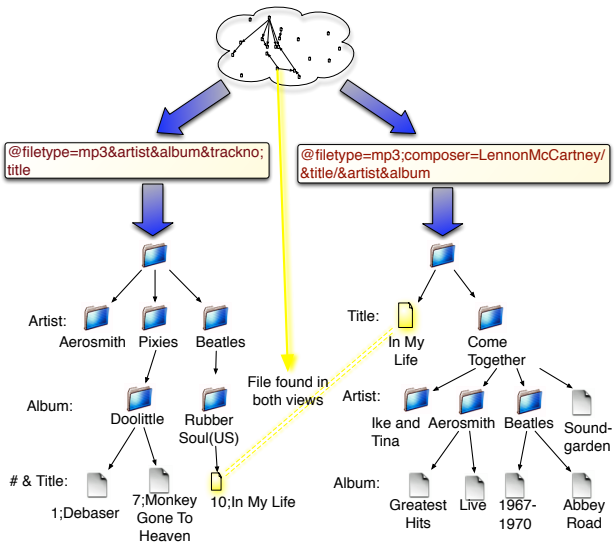

Figure 2: *Two parallel virtual directory hierarchies containing a common file*

specifies views, results may be returned either as tuples or hierarchically. Tuples contain values from multiple fields concatenated into a single entry. For example, song file results are listed using the well known fields: "Artist Name", "Album Name", "Track Number" and "Track Title", but with QUASAR, we may change the ordering or presence of each value. This advantage provides for flexible naming. Hierarchical results present values for each field per subdirectory level. We place together results that share common values into virtual subdirectories. For example, as depicted on the right side of figure 2, we group query results of songs sharing a common title in the first group of subdirectories. The organization follows down the tree with results sharing artists, and finally, with the albums on which each song appears.

QUASAR's generalized approach to querying overcomes the rigid nature of applications' search interfaces. Suppose that a new category of attribute may become desirable for music search. For example, a music critic wants a collection of tracks produced or engineered by a famous producer in a particular year, and tags music files for that purpose. While GUI applications may need to be updated to accommodate the new field, the underlying search interface provided through QUASAR remains the same, and better yet, advanced users who directly use the query language can immediately take advantage of new categories.

4 Use of QUASAR

Our file systems query language solution, QUASAR, gives users the power to conjoin several different styles of search and view definition within a single, compact query. When confronted with the full breadth of fea-

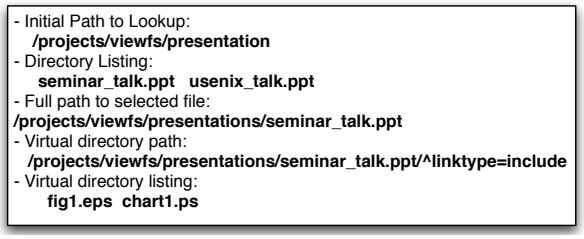

Figure 3: *Steps to create a simple virtual directory using relationship-based search.*

tures, a user may consider QUASAR very complex. As with any new technology, there will be a learning curve. However, we do not expect a steep learning curve, relative to other similar languages. First of all, we have designed the language such that the totality of its features need not be used in order to achieve results. For example, let us consider the baseline for QUASAR's usage where it appears like a POSIX file system. Here, path separators traverse over "containment" relationships, utilizing the "name" field on links, while other search features are inactive. Secondly, storing the current working directory hides much of QUASAR's complexity by allowing users to avoid retyping long strings and to gradually build and refine queries. Additionally, we support the use of utilities to provide aliases for common query fragments, so that users may easily compose and expand their queries.

Figure 3 presents an example of a virtual directory, introducing a simple use of a **relationship-based search**: a new search technique we introduce through QUASAR. The first couple of steps show paths that look like POSIX, and in fact, these paths behave like standard directory traversals, as we described in the baseline. The addition of our syntax, **/^linktype=includes**, creates a virtual directory of all files that are included by the preceding file, provided we set traversal to our default operation. However, we are not limited to "included" files, as we may consider other relationships as well. Also, more advanced relationship-based QUASAR query operations can perform refinements based on attributes found on link target or source files.

To further demonstrate QUASAR's backward compatibility with existing search methods, we present how to use the language as one would use a search engine. A search for "viewfs seminar presentation" would translate into **@viewfs;seminar;presentation**. The @ character tells the system to search the entire file system for matches, though the operation may be set as a default for search, and the character omitted. For this example, we have configured a default search field of "keyword", so that the field name may be omitted from each term.

Once a user becomes familiar with the two query styles presented above, she or he may combine them for added results. Our example using three terms should produce a list of results that match these terms. Selecting search results in QUASAR is exactly like choosing a directory entry in POSIX. We may append the file name of our chosen result, in this case a PowerPoint presentation file, to the query string, with the POSIX separator character. Then, if we would like to see any files included by the selected presentation, we may append our previously mentioned relationship-based search operation string. The final query appears as: **@viewfs;seminar;presentation/seminar_talk.ppt/^linktype=include**

Now, we would like to see what has been included by *all* result files from a keyword search. The QUASAR query to accomplish this task is similar to the one above: **@viewfs;seminar;presentation/^linktype=include**
In this case, QUASAR traverses all "include" links as a group, from each result that matches the query terms. This method works much like XML node traversal using XPath. The combining of multiple search techniques, in addition to being backwards compatible with POSIX, endows QUASAR with advantages over other proposed semantic file systems. QUASAR additionally supports search terms with field names, featured in some of the semantic file systems, and available in Spotlight's "smart folder" interface in a limited fashion. Here is the term query from above with field names added:
@project=viewfs;filetype=presentation;topic=seminar*

5 Location-Based Search

In answer to the problem of locating particular files out of a large system, we present the strategy of searching by location. Consider web-based keyword search, where we often encounter an unmanageable numbers of results. If we reduce the result set to contain only the results within a particular subsection of the web, we can limit the number of results to a more manageable number. Within file systems, one way to limit search to location is to search within subtrees of a global directory hierarchy. However, it is not realistic to confine oneself to a single subtree, even when using symlinks to other subtrees (when necessary). Over time, what was originally a manageable subtree will grow to the unmanageable proportions we initially wished to avoid. We do know that subtrees are defined by links within the file system. Moreover, given our metadata model, we find links for relationships other than simply containment.

One such relationship is *provenance*: by using links derived from provenance relationships, we may establish locations within file systems, where files are co-located when they are "descended" from an earlier file. For instance, an absent-minded oceanographer wishes to find

*we assume we can tag all presentations under this field, as opposed to using a more specific field for file format

his presentation slides on wave systems. Keyword search produces a large number of results, but he is not sure which file is correct, and wants to avoid opening each one. However, he recalls that he has derived several images, included by the presentation files, from a script used to model waves. So, if he locates the script via browsing and keyword search, he may reduce the list of presentations to the correct few. To make searches like this simple, QUASAR features a single operation that defines a search location based on link distance. When such a query is stored as a current working directory, subsequent queries may naturally be confined to this predefined space.

Unlike relying on term search alone, where result sizes grow to be unmanageable with larger file systems, location-based search has the advantage of being consistently based within a well-defined set of files, no matter how large the system. The link distance is an easy-to-control search parameter that we have built into QUASAR's syntax, which can expand and contract result set sizes. Moreover, the *virtual root* of these searches is always flexible, as well.

6 Implementation

To experiment with the use of QUASAR, we have developed a prototype implementation that runs in user space. Figure 4 shows the interaction of its components. The FUSE framework allows us to mount the file system within a system's global namespace in order to conduct our experiments. Paths presented to the mountpoint that contain QUASAR syntax are evaluated by a query engine in the user file system. We derive our query language parser from Flex and Bison. Our parser produces a query-plan data structure that we process against our indices. The user file system has its own custom metadata store and indices. We keep these in system RAM, using binary search trees that facilitate future range queries.

For our experimental metadata, we have imported an iTunes library (citation). To experiment with links, we have utilized playlists that reference other files. The focus of our experiments was to verify that each query operation returned search results correctly, within a reasonable response time. Under both a Powerbook G4 (MacOSX) and AMD 64 server (Linux 2.6.x) deployments, response times for queries were consistently under 10 ms of system and user time. Our previous experience with FUSE indicated that there is considerable overhead consisting of message passing from kernel to user space. However, our experiments with the prototype were geared toward usability and proof-of-concept as opposed to performance. We have successfully demonstrated our query operations for keyword search, link traversal, link distance searching, matching files based

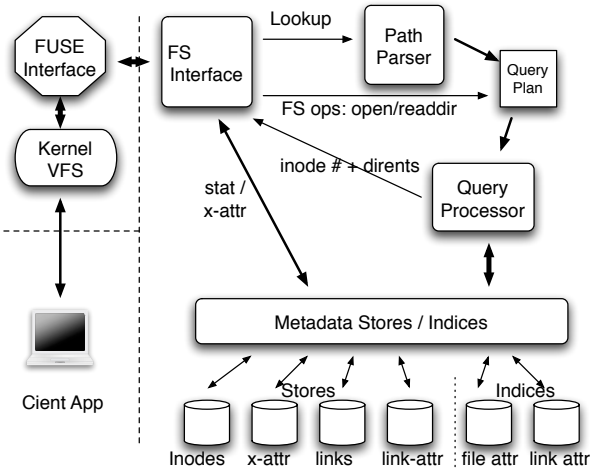

Figure 4: *The prototype software architecture with custom metadata storage.*

on link target (child) attributes and matching based on link source (parent) attributes. In one sample search, we have determined which of the playlists link to songs of a particular artist or genre. In another, we found songs that are common to two playlists. Such a query combines: a simple traversal to locate the first playlist; a second single traversal to target song files; refinement to match particular attributes on the song files; and additional refinement through matching songs with the second playlist as a parent file. We successfully demonstrated a link distance search: we identified a parent file that contains all the playlists, set the search distance to two links, and refined to match song files by particular artists.

Additionally, we successfully listed files by names that were derived from an arbitrary selection of attributes, in a shell environment, using the common *ls* command. We have combined the use of "current working directory" with our QUASAR test environment to browse virtual directory hierarchies, and also to refine queries. For the latter, we performed two test cases. In the first, we set a working directory context of particular "Genre" and "Artists", and refined our results through listing particular albums. In the second, we set the context to only contain our playlist files, and continued the search by traversing links to a virtual collection of song files from those playlists, and by matching specified criteria.

Existing POSIX symbolic linking gives us a pre-existing framework for storing queries. Our prototype can store a query as the symbolic link target, and opening the link evaluates the query. Additionally, when using our MacOSX test deployment, we placed some queries in symbolic link targets elsewhere in the HFS. Opening the links in Finder allowed us to browse the virtual hierarchies exactly as one would browse regular directories. We have also tested the same functionality under the Gnome file browser for LINUX.

7 Metadata Typing

"Date range" is a common query predicate that we expect a search system to answer. Moreover, there are other quantities and values that should be available for range queries. Range queries are interesting: their execution depends on the ordering of values, which requires that metadata must be properly "typed" so that it may be ordered either numerically or lexicographically. Traditional FS metadata fields, such as dates and sizes, are stored as integers, and thus should be treated as such for indexing and ordering.

However, to our knowledge, file systems search tools that utilize extended attributes for metadata cannot properly handle numeric ranges, because extended attributes are treated as strings. For instance, Beagle uses Lucene to index its extended attribute metadata. Dates and numeric values must be converted to strings that mimic numeric ordering by prepending 0's, which is obviously problematic. In contrast, Spotlight handles traditional numeric file system metadata as well as some common well-known attributes in a proper fashion. Unfortunately, instead of a flexible, extensible approach, Apple supports fields based only on its applications' needs.

We have identified two alternatives that should handle typing for extended attributes, and will present ramifications for each. First is explicit typing, where we may introduce a system call to register types for attribute fields. The system must maintain a database of types. One question that arises from this scheme is how to handle conflicting type registrations: for instance, whether the use of polymorphic fields should be considered. The second alternative is to utilize the parser to determine a value's type at both indexing and query times. The advantages are that first, applications can determine the type with which to treat a field, and use that field independently of other applications; second, terms should continue to match attributes, regardless of type. This approach could be problematic if an application requires a field to be strictly alphanumeric, and mixed values do not order properly.

8 Conlcusion and Future Work

We have presented a need for search capabilities within file systems. To reconcile issues of inconsistent approaches outside of the file system, search capability must be incorporated into the kernel interface. A query language supports the combination of multiple search techniques and view specification. Through QUASAR and our prototype implementation, we can see that the search techniques we have proposed are indeed feasible.

Future performance evaluation of a single-host system running QUASAR shall be conducted through an in-kernel prototype. Using this implementation, we may focus on other related elements of the work, including: implementing the semantics for addition and modification of metadata; our proposed typing schemes; query results caching; and an interface to a service that receives updated query results. To formally model the query processing, we are developing a language calculus. Furthermore, we hope to integrate the interface with a distributed indexing framework that will support search over distributed storage clusters, so that we may experiment with queries at a much larger scale.

Acknowledgments

This work was supported in part by the Department of Energy under award DE-FC02-06ER25768, by Lawrence Livermore National Laboratory, and by the industrial sponsors of the Storage Systems Research Center at the University of California, Santa Cruz. We thank the members of the SSRC for their feedback.

References

[1] AMES, A., BOBB, N., BRANDT, S. A., HIATT, A., MALTZAHN, C., MILLER, E. L., NEEMAN, A., AND TUTEJA, D. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies* (Monterey, CA, Apr. 2005).

[2] AMES, S., BOBB, N., GREENAN, K. M., HOFMANN, O. S., STORER, M. W., MALTZAHN, C., MILLER, E. L., AND BRANDT, S. A. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies* (College Park, MD, May 2006), IEEE.

[3] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, JR., J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)* (Oct. 1991), ACM, pp. 16–25.

[4] MARCHIORI, M., AND QUIN, L. W3c xml query (xquery). http://www.w3.org/XML/Query/, 2007.

[5] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track* (2006), pp. 43–56.

[6] PADIOLEAU, Y., AND RIDOUX, O. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003), pp. 99–112.

[7] PRUD'HOMMEAUX, E., AND SEABORNE, A. Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query/, 2007.

[8] SHAH, S., SOULES, C. A. N., GANGER, G. R., AND NOBLE, B. D. Using provenance to aid in personal file search. In *Proceedings of the 2007 USENIX Annual Technical Conference* (June 2007), pp. 171–184.

[9] SOULES, C. A. N., AND GANGER, G. R. Connections: using context to enhance file search. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM Press, pp. 119–132.