Innovation, Sciences et Développement économique Canada
Office de la Propriété Intellectuelle du Canada

Innovation, Science and Economic Development Canada
Canadian Intellectual Property Office

CA 3028785 C 2019/12/31

(11)(21) 3 028 785

(12) BREVET CANADIEN
CANADIAN PATENT

(13) C

(86) Date de dépôt PCT/PCT Filing Date: 2017/07/20
(87) Date publication PCT/PCT Publication Date: 2018/02/22
(45) Date de délivrance/Issue Date: 2019/12/31
(85) Entrée phase nationale/National Entry: 2018/12/19
(86) N° demande PCT/PCT Application No.: US 2017/043006
(87) N° publication PCT/PCT Publication No.: 2018/034785
(30) Priorité/Priority: 2016/08/16 (US15/238,008)

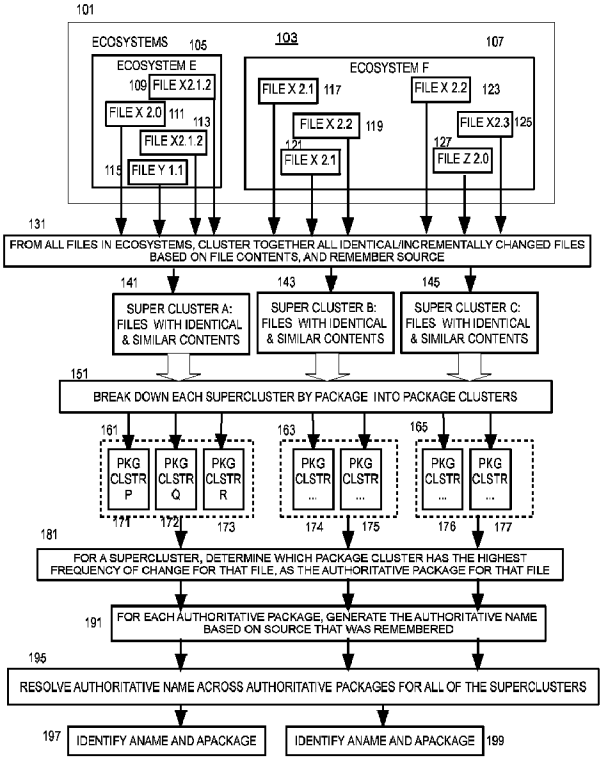
(51) Cl.Int./Int.Cl. *G06F 16/906* (2019.01), *G06F 16/17* (2019.01), *G06F 8/70* (2018.01)

(72) Inventeurs/Inventors:
FOX, BRIAN, US;
MAYHEW, BRUCE, US;
WHITEHOUSE, JAMIE, US;
MAHMUD, GAZI, US

(73) Propriétaire/Owner:
SONATYPE, INC., US

(74) Agent: OYEN WIGGS GREEN & MUTALA LLP

(54) Titre : METHODE ET SYSTEME D'ANALYSE DE NOM DOCUMENTE DE L'ORIGINE D'UN FICHIER
(54) Title: METHOD AND SYSTEM FOR AUTHORITATIVE NAME ANALYSIS OF ORIGIN OF A FILE



(57) Abrégé/Abstract:
A computer system, method, or non-transitory computer-readable medium provides an authoritative name source for files within an ecosystem (103, 105, 107). Files (117-127) in the ecosystem (105, 107) which have identical contends and similar contents to

(11)(21) 3 028 785
(13) C

(57) Abrégé(suite)/Abstract(continued):
each other are merged (131) into the same supercluster (141, 143, 145), to capture possibly incremental changes to the files over time in one of the superclusters. For each supercluster (161, 163, 165) which has files with identical and similar contents, the supercluster is broken down (151) into package clusters (171-177), based on packages to which the files belong, each of the package clusters has the files from a same package. The package cluster which has most change frequency across versions (181), is identified as the authoritative package. The authoritative name for the files is resolved, based on the authoritative packages that are determined, across the plurality of superclusters which have files with identical and similar contents, and the authoritative name is generated (191). Any authoritative name collision is resolved (195).

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
22 February 2018 (22.02.2018)

WIPO | PCT

(10) International Publication Number
WO 2018/034785 A1

(51) International Patent Classification:
G06F 17/30 (2006.01) *H04L 29/12* (2006.01)

(21) International Application Number:
PCT/US2017/043006

(22) International Filing Date:
20 July 2017 (20.07.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
15/238,008 16 August 2016 (16.08.2016) US

(71) Applicant: **SONATYPE, INC.** [US/US]; 8161 Maple Lawn Blvd., Suite 250, Fulton, MD 20759 (US).

(72) Inventors: **FOX, Brian**; c/o Sonatype, Inc., 8161 Maple Lawn Blvd., Suite 250, Fulton, MD 20759 (US). **MAYHEW, Bruce**; c/o Sonatype, Inc., 8161 Maple Lawn Blvd., Suite 250, Fulton, MD 20759 (US). **WHITEHOUSE, Jamie**; c/o Sonatype, Inc., 8161 Maple Lawn Blvd., Suite 250, Fulton, MD 20759 (US). **MAHMUD, Gazi**; c/o Sonatype, Inc., 8161 Maple Lawn Blvd., Suite 250, Fulton, MD 20759 (US).

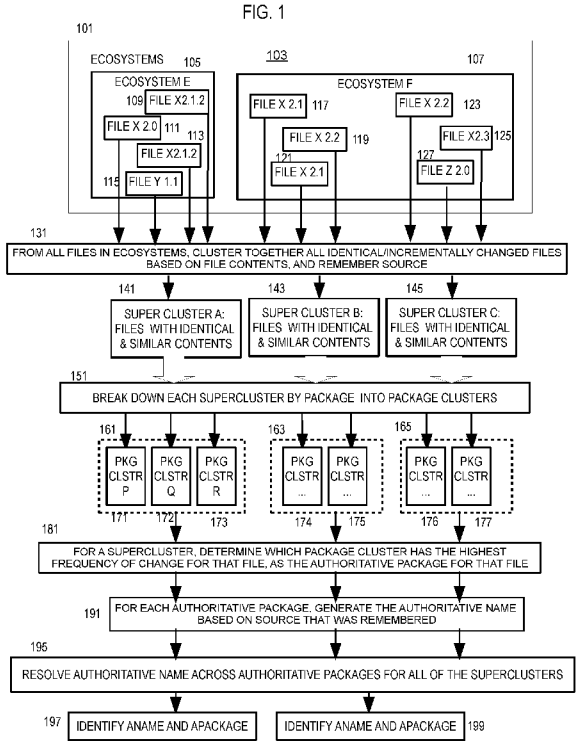
(74) Agent: **POSZ, David, G.** et al.; Posz Law Group, PLC, 12040 South Lakes Drive, Suite 101, Reston, VA 20191 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

(54) Title: METHOD AND SYSTEM FOR AUTHORITATIVE NAME ANALYSIS

FIG. 1



(57) Abstract: A computer system, method, or non-transitory computer-readable medium provides an authoritative name source for files within an ecosystem (103, 105, 107). Files (117-127) in the ecosystem (105, 107) which have identical contends and similar contents to each other are merged (131) into the same supercluster (141, 143, 145), to capture possibly incremental changes to the files over time in one of the superclusters. For each supercluster (161, 163, 165) which has files with identical and similar contents, the supercluster is broken down (151) into package clusters (171-177), based on packages to which the files belong, each of the package clusters has the files from a same package. The package cluster which has most change frequency across versions (181), is identified as the authoritative package. The authoritative name for the files is resolved, based on the authoritative packages that are determined, across the plurality of superclusters which have files with identical and similar contents, and the authoritative name is generated (191). Any authoritative name collision is resolved (195).

WO 2018/034785 A1

[Continued on next page]

WO 2018/034785 A1

Published:
— *with international search report (Art. 21(3))*

METHOD AND SYSTEM FOR AUTHORITATIVE NAME ANALYSIS OF ORIGIN OF A FILE

TECHNICAL FIELD

[0001] The technical field relates in general to software development, and more particularly to authoritatively identifying files and packages used in software development.

BACKGROUND

[0002] In today's software development field, software components that are used in connection with an application are usually written by others, and are used and re-used, possibly with changes. This trend is increasing.

[0003] A package is conventionally used to organize a set of related components intended for use by a particular application. The package may contain, by way of example, content needed to reference an external library; source code, binaries, executables, classes, and interfaces; identifying information and a manifest that identifies requirements if the package; scripts, other packages, and related items that run when the package is installed, de-installed, and/or used by a software application. A developer of a file or package typically will publish the package with that file to a repository so that the package can be shared and re-used, such as by other developers.

[0004] The available tools for software development makes it easy to use packages and software components, but these tools do not make it easy to understand where the package or the files in the package originated. Compounding the lack of visibility and understanding is the fact that packages and files in packages tend to depend on other components and/or packages and files which may themselves be used or re-used, possibly with changes.

[0005] A software repository is a known technique to provide developers with a convenient collection of re-used and re-usable packages and software components, where the package and/or component in the software repository may be a copy or specialized version and/or may incorporate other components and/or packages such as by dependency. A conventional repository manager can be used as a central point of storage and exchange for packages and software component usage. Package management systems are available which can provide tools

CA 3028785 2019-05-22

to automate distribution of packages, can maintain information about dependencies and version information, and can work with software repositories. For a single ecosystem which is a collection of software repositories and/or support systems all of which store interrelated files, there can be thousands of distinct software packages with their attendant files.

[0006] Nevertheless, the problem with visibility and understanding of packages and the software components remains. There may be numerous files in a repository which are essentially the same component but which might have minor variations in content and/or name; any of these components might have originated elsewhere in the ecosystem. Using a repository manager or a package manager to identify the software repository where an application, package or component is stored or managed, does not necessarily correctly identify where that package or component actually originated and does not necessarily identify the name of the original file or the package where the file originated.

[0007] In particular, with all of the sharing of files via packages, it remains problematic to correctly locate the original name and version of a file which appears in different packages, in combination with the original package which originated that component.

SUMMARY

[0008] Accordingly, one or more embodiments provide a computer system which provides an authoritative name source for files within an ecosystem. The computer system includes at least one processor is configured to perform the following.

[0009] The files in the ecosystem are clustered into a plurality of superclusters, in which the files in each supercluster of the plurality of superclusters have identical contents. Then, of the files in the ecosystem which are clustered into the plurality of superclusters, which of the files have similar contents to each other are determined, and the files which have similar contents to each other are merged into the same supercluster, to capture possibly incremental changes to the files over time in one of the superclusters which has the files with identical contents and similar contents.

[0010] For each supercluster which has the files with identical and similar contents, the following is performed. Each supercluster is broken down into package clusters, based on

packages to which the files belong, so that each of the package clusters has the files (that is, the files with identical and similar contents (probably due to incremental changes) from a same package. Which of the package clusters has most change frequency across versions is determined, as the authoritative package.

[0011] Then, an authoritative name for the files is resolved, based on the authoritative packages that are determined, across the plurality of superclusters which have files with identical and similar contents, and generating the authoritative name. Any authoritative name collision is resolved.

[0012] In an embodiment, a hash of file contents of each of the plurality of files is used to determine which of the files have identical contents, and the clustering of the files in the superclusters with identical contents is determined based on the hashes of file contents.

[0013] In an embodiment, for each supercluster, the each supercluster includes an indication of a source within the ecosystem for each file of the files, and the breaking down into the package clusters is based on the indication of the source of the each file.

[0014] In an embodiment, an abstract syntax tree of file contents of each of the plurality of files is prepared, and the abstract syntax trees are used to determine whether contents of files are similar, and the determining of files with similar contents into the same supercluster is based on the abstract syntax tree of file contents,

[0015] In an embodiment, for each supercluster, the each supercluster includes an indication of a source within the ecosystem for each file of the files.

[0016] In yet another embodiment, breaking each supercluster down into package clusters includes generating package names and versions for members of the supercluster; aggregating package clusters on package name and counting unique changes to the package cluster; and determining, for each package cluster, the most change frequency among the package clusters.

[0017] In still another embodiment, the authoritative name is resolved by merging authoritative package outliers into groups which are merged into package cluster based on file name; and the resolved authoritative name and source are identified.

[0018] In yet a further embodiment, whether the authoritative name source is correct is determined based on the ecosystem in which authoritative name source originated.

[0019] In still another embodiment, the ecosystem, in which the files are located which are clustered, includes at least one software repository in which the files are located.

[0020] A further embodiment is a computer-implemented method, according to one or more of the above embodiments.

[0021] Still another embodiment is a non-transitory computer-readable medium comprising instructions for execution by a computer, the instructions including a computer-implemented method as above, the instructions for implementing the method in a processor.

[0022] One, or a combination of more than one, or all, of the above embodiments, can be combined and provided as a single embodiment.

[0023]

BRIEF DESCRIPTION OF THE DRAWINGS

[0024] The accompanying figures, where like reference numerals refer to identical or functionally similar elements and which together with the detailed description below are incorporated in and form part of the specification, serve to further illustrate various exemplary embodiments and to explain various principles and advantages in accordance with the embodiments.

[0025] FIG. 1 is a data flow diagram illustrating flow in a computer system for authoritative name analysis;

[0026] FIG. 2 is a block diagram illustrating a supercluster of files with identical and incrementally changed contents, broken down into package clusters;

[0027] FIG. 3 is a simplified file listing of files by hash across multiple versions in a package cluster;

[0028] FIG. 4 is an overall flow chart illustrating a procedure for authoritative name analysis;

[0029] FIG. 5 is a block diagram illustrating relevant portions of a computer system;

[0030] FIG. 6 to FIG. 8 collectively are a flow chart illustrating a procedure to provide an authoritative name analysis.

DETAILED DESCRIPTION

[0031] I. Introduction

[0032] In overview, the present disclosure concerns observing a file in a packaged application and indicating what is known about said file; more particularly uniquely identifying a correct name for a file within an ecosystem(s) which has numerous files, and/or identifying the actual origin of the file within the ecosystem, in which a package which may be associated with the packaged application comprises different self-contained software components which are stored in the ecosystem, the ecosystem may include one or more software repositories which support different packages, and the software components and packages are used and re-used by developers. Even more particularly, the present disclosure concerns attempting to identify, from all of the files in that ecosystem, a package in which a file changes or is a new version each time the package is released, which package will be considered to be the authoritative source of the file; the authoritative name of the file can be identified from the package which is the authoritative source. The present disclosure builds on a hypothesis that an original author of a file will not push a new package of that file unless that file is changed; whereas a developer that merely uses the package with that file will likely not change that file in every release of a package.

[0033] More particularly, various inventive concepts and principles are embodied in systems, devices, and methods therein for an authoritative name analysis that definitely determines, for a given file, the true origin of the file (a so-called authoritative package), independent of how the file is packaged in an ecosystem, and which determines the authoritative name of the file. The authoritative name comprises the name of the file and a canonical version.

[0034] The approach discussed herein to component identification can provide superior support for component identification, e.g., Java component identification in a programming language sold under the trade-mark or trade name JAVA, compared to available techniques. The strength of this approach can be enhanced by use of a few known conventions A-D below in the Maven/Java ecosystem which are noted at the outset in a software project management tool sold under the trade-mark MAVEN:

[0035] A. Clear distinction between release and development versions:

[0036] Maven includes the concept of a SNAPSHOT version that is used for development. It also includes numerous tools for release automation to create release binaries.

[0037] B. Large Centralized Repository:

[0038] These release binaries are typically published to the Central Repository, run and sponsored by Sonatype. The central repository contains all releases since the beginning of Maven usage as immutable binaries.

[0039] C. Standardized Rich Metadata Format:

[0040] The conventional POM format used for the pom.xml file for component metadata is well standardized and therefore suitable for automated parsing. It is also comprehensive enough to allow for detailed component and dependency information.

[0041] D. Typically Immutable Components :

[0042] The immutable release binaries in the Central Repository and other repositories are typically consumed and used by other applications, without being modified. While modifications from simple file renames, recompilation, tree shaking, shading and others are possible, they are not the common use case.

[0043] The above conventions A-D might be expanded to other repositories and/or ecosystems.

[0044] II. Problem Presentation and Observations

[0045] A. Problem Statement

[0046] Previously, in software repositories, partial matching and recognizing of third party system components has been performed. This approach was primarily Java-specific and works well for Java environments. In other environments and/or languages, such as offered under the trade-marks or trade names NuGet, npm, Python, Ruby, etc. there is a package vs. component

problem. Java uses JARs, and therefore the software repositories use JARs and are very well-behaved. Nevertheless, other languages and environments use files which contain components, e.g., DLL files and others. The ecosystems tend to be massively not well behaved. For example, in a NPM repository, developers are not using the proper build format for using the library offered under the trade-mark JQuery (which is used as a convenient example) but instead incorporate their own version of JQuery. Complicating the problem in this example is that JQuery (as an example) changes in every release.

[0047] There are other problems in attempting to identify a name and authoritative source of a file, caused by something even as simple as files not being named properly by developers. This complicates identification of the file name and/or file type.

[0048] All of this can occur in a Big Data environment with thousands of files.

[0049] Further, simply matching the text of an identical file name is easy, but confusing because a filename may not be the commonly recognized name.

[0050] As a simple example, the following steps will usually not identify the authoritative name:

[0051] 1. recognize a javascript file (this is a routine technique)

[0052] 2. determine the file name.

[0053] Nevertheless, it is difficult for the computer to determine whether the file is a form of some real name as known to humans.

[0054] Using a timeline so as to identify the first or earliest component in a software repository ordinarily is not helpful, because a component may be added to a software repository asynchronously, and/or without regard to whether the component is the ultimate source.

[0055] Furthermore, as the usage of repositories and repository managers is adopted by other ecosystems, less mature or suitable usage patterns emerge and have to be considered for component identification and software supply chain management efforts.

[0056] The typical enterprise usage also includes component usage that spans ecosystems. In fact, this behavior is the norm. Consequently, it is likely that a particular file did not originate in the same ecosystem in which it is being used. An effect is that it is probably erroneous to simply identify the earliest use of that file, or the earliest project corresponding to that file.

[0057] All of these technical advances lead to numerous problems with precise component identification, which can cause problems with, for example, software supply chain management.

[0058] B. Problem Example -- JavaScript and JQuery

[0059] Reference is now made to a real example to showcase some aspects of the problem.

[0060] JQuery is a very popular JavaScript framework used for web application development. Even the official download page mentions numerous options and formats:

[0061] • Direct download of the compressed (=minified) file for production usage

[0062] • Direct download of the uncompressed file for development

[0063] • Download with npm

[0064] • Download with bower

[0065] • Download from/Reference to numerous CDN hosting options

[0066] Of course, there are numerous older releases of JQuery with three major versions currently available. There are also subprojects such as jquery mobile and jquery colors available. One will appreciate, even for the core jquery library, there are numerous ways to obtain the code and the library.

[0067] Now consider a typical usage. As a library sold under the trade-mark or trade name JavaScript, JQuery designed for client side usage in a web browser is used in many web applications. These are written in various different development stacks sold under trade-marks or trade names such as Node.js, .Net and Java. Respectively, jquery files can be found in many web applications in the repositories for these stacks: not only in Maven repositories such as the Central Repository, in repositories sold under the trade-marks or trade names NuGet used by .Net developers such as the repository offered under the trade-mark NugetGallery, in a repository offered under the trade-mark or trade name npmjs.org package managers offered under the trade-marks and/or trade names used by Node.js developers, but also in many others including internal repositories using the different formats. As a result, a developer might for example find a Java WAR file that contains a jquery.js file in the a js or other folder inside the WAR archive.

[0068] As another complication, the libraries by themselves are often also packaged in the specific repository format to ease the usage. For example, JQuery is distributed via, among many others:

[0069] • npm https://www.npmjs dot com/package/jquery

[0070] • NuGet https://www.nuget dot org/packages/jQuery

[0071] • Maven http://repo1.maven dot org/maven2/org/webjars/jquery/

[0072] These redistributions are often not managed by the actual JQuery project itself. JQuery is also used by many other libraries, written in JavaScript or any other stacks, as a dependency that provides functionality as a foundation for further features and therefore is embedded in the relevant packages.

[0073] Keeping this in mind, a current analysis finds that a jquery.js file is found in roughly 72,000 packages across the various packaging systems and repositories in one ecosystem. The total number of distributions is even higher, and the number of packages continues to increase.

[0074] C. Challenges

[0075] The sheer amount of packages to analyze and the resulting data that needs to be sifted itself poses a challenge.

[0076] In addition, there other important challenges which include the following.

[0077] --How can the system figure out that the file jquery.js occurs in a specific package merely because it is simply used in that package?

[0078] --How can the system determine the specific version of jquery which this package contains? And how can the system differentiate this package's version of jquery from an original jquery distribution itself in a specific packaging format?

[0079] --How can the system recognize that some file is actually the original jquery despite jquery having been included in 72,000 different packages, so that the system can provide the right metadata which corresponds to the original jquery?

[0080] Solving these questions is necessary to be able to correctly associate the right metadata to the right packages, so as to provide the correct information downstream. For example, a correct identification of components within other components and their distribution supports the creation of a correct bill of materials of an application or the performance supply chain management without wasted efforts.

[0081] These challenges affect all software ecosystems and software packaging systems. Accordingly, solving the identification problem is not at all isolated to JavaScript usage.

[0082] D. State of the Conventional Art

[0083] There are problems with current approaches. What is not working today, is that the identification problem has not been solved. Without solving the identification problem in detail, any further efforts are flawed. And that is exactly where the conventional art is.

[0084] Conventional systems which assert JavaScript support often mean that *.js files are identified as JavaScript and not much more beyond.

[0085] Some conventional systems can identify jquery.js as JQuery, but fail when the file is renamed. Some conventional systems fail when JQuery is embedded in other files nested in archives. Some conventional systems fail to identify the version of JQuery.

[0086] Although JavaScript support is regularly marketed as a feature of conventional products, the inventors have not observed truly working support necessary for efficiently and correctly identifying files and packages. In fact, similar issues are found in actual use of conventional Java/Maven support products.

[0087] **III. Aspects of the Approach**

[0088] Further in accordance with exemplary embodiments, there is provided a method and system that resolve the issues mentioned above.

[0089] As further discussed herein below, various inventive principles and combinations thereof are advantageously employed to allow determination of the authoritative name of a package.

[0090] A. Concept Summary

[0091] The overall concept is to identify the authoritative source of a file, and the authoritative name of a file. The point is to be able to identify that a component is a form of something authoritative, which is known to the developers. As a brief summary, the concept uses the following steps, in an ecosystem which has large numbers of files:

[0092] 1. Cluster the files in an ecosystem(s) into superclusters of files, where each supercluster has identical and partially matching components/files, while retaining an indication of the package source of each component/file. Each supercluster thus has all of the different packages which use and re-use the basically identical file (as that file incrementally evolves from time-to-time).

[0093] 2. Within each supercluster, cluster the same packages together and look for the smallest package cluster, which is the package cluster in which the file incrementally evolves with the highest frequency. Consider the package cluster with the highest frequency of change to identify the package which is the authoritative source of a file, and from there ascertain the authoritative name of the file.

[0094] This solution is totally counter-intuitive to the usual use of clustering of files which relies on the highest confidence (largest) cluster instead of the smallest and thus ordinarily the lowest confidence cluster.

[0095] B. Authoritative Name and Authoritative Package Concept

[0096] A goal of the Authoritative Name analysis herein is to correctly determine the authoritative name of a package. The authoritative name is the identifier used for a package that ties the package to the original upstream release of a component and therefore the correct metadata.

[0097] The solution to the problem discussed in the above section is correct and precise component identification. The redistribution and nesting of components is handled by determining which packages were redistributed within another package, as distinguished from real distribution based on an original upstream project. As an advantage, attachment of the correct metadata to the component is possible when this is achieved. The process and system discussed herein makes this distinction possible.

[0098] There are several advantages of this approach, including the following examples. The concept behind the authoritative name identification can be independent of the packaging system(s) by which components are distributed. The concept of authoritative name identification enables a new quality of component identification in JavaScript and other ecosystems. Authoritative name identification can enable further improvements, such as to the Java/Maven support and others.

[0099] Consider, as a simple example, the JavaScript library Clipper has a dependency to jquery. The authoritative name analysis of any Clipper package should result in an authoritative name of "Clipper"; the authoritative name analysis of any "ApplicationX" that uses Clipper

should result in an authoritative name of "ApplicationX". However, neither the Clipper package nor ApplicationX can result in an authoritative name of "jquery".

[00100] On the other hand, authoritative name analysis of any redistribution of JQuery in npmjs.org, the NuGet Gallery or the Central Repository, should result in the authoritative name of "jquery".

[00101] Authoritative name analysis of any usage of the identical jquery file, or even a renamed or custom minified copy of the original jquery.js, within an application, should result in the identification of this component within an application as "jquery" -- not "Clipper" or any other authoritative name.

[00102] From a developer's perspective, the authoritative name analysis can work off just the distributable application archive or other specified group of files. The authoritative name analysis does not require any build time declaration or other context specific information to provide correct results. Consequently, a developer can provide an authoritative name analysis even of a binary running in a production environment.

[00103] All contents of an application distributable can be extracted and precisely identified for authoritative name and version. As a result, a developer can provide correct association of metadata, and a creation of a complete bill of materials or other usage of the metadata.

[00104] The Authoritative Name analysis discussed herein can provide support for the JavaScript ecosystem and/or other ecosystems.

[00105] The nature of the authoritative name analysis includes large data mining and analysis efforts because the ecosystems tend to have thousands of files.

[00106] IV. Details of Concepts and Variations

[00107] The following example and some variations are well suited to demonstrate various implementation details and advantages.

[00108] A. Overview

[00109] The following is an overview of the steps which can be followed to provide an authoritative name analysis. This example mentions Javascript, but it should be noted that the

authoritative name analysis is intended to be applied and adapted (if necessary) to non-Javascript files as well.

[00110] 1. Identify the contents of every file in the ecosystem(s) -- Collect the hash of every Javascript file across the various ecosystems

[00111] 2. Determine all files which have identical contents -- Create file hash clusters (FHC) for all Javascript files, in which each cluster has only files with identical contents based on, e.g., hash of file contents

[00112] 3. Determine all files which have contents which are incrementally changed from the identical files -- Aggregate/enlarge file hash clusters, e.g., based on Abstract Syntax Tree ("AST") similarity

[00113] 4. For a cluster of identical/incrementally changed files, determine the packages the files belong to, further break those packages down by versions, and identify the package with the "most change" frequency for that file -- Create package clusters from file hash clusters

[00114] 5. Pull any outlying files back into the package version group in which they belong -- Create package version groups from the package cluster based on authoritative package versions

[00115] 6. Resolve authoritative name collisions among the clusters of files -- Generate AName from authoritative package version groups

[00116] The purpose of steps 1-3 is to find all of the different versions of a file and partially modified files from those versions, for example, to put all of jquery files into a supercluster. The purpose of the next steps 4-6 is to sort the supercluster out, to determine the versions, the packages, the authoritative package and the authoritative name, for the file. The steps can handle variant cases to achieve a more correct result.

[00117] All of the above are discussed in more detail in the following section in connection with FIG. 1, a data flow diagram illustrating flow in a computer system for authoritative name analysis. As illustrated in FIG. 1, there are files in packages (not illustrated) in one or more ecosystems 103, here represented by Ecosystem 105 and ecosystem F 107. Files in ecosystem E 105 include File X 2.1.2 (plural copies), File X 2.0, and File Y 1.1 109-115. Files in ecosystem F 107 includes File X 2.1 (plural copies), File X 2.2 (plural copies), File X 2.3, and File Z 2.0

117-127. The ecosystem(s) have only those files of which the ecosystem is aware, and accordingly, there are gaps in versions and package releases in the ecosystem. The files are identified 101, and then from preferably all of the files in the ecosystems 103, the analysis clusters 131 together, into superclusters, all identical and incrementally changed files based on file contents, and remembers the source where the files originated. Here, each of supercluster A, B and C 141, 143, 145 has different files, and the files in one supercluster have identical and sufficiently similar (i.e., incrementally changed) contents to each other.

[00118] Then each supercluster 161, 163, 165 is broken down by package into package clusters171-177, in which each package cluster has files from the same package. Then, for each supercluster, the analysis determines 181 which package cluster has the highest frequency of change for that file, which is the authoritative package for that file. For the authoritative package, the analysis generates 191 the authoritative name based on the source that was remembered. The analysis resolves 195 the authoritative name across authoritative packages for all of the superclusters in view of the possibility of collisions among authoritative names, possibility resulting in fewer authoritative names and authoritative packages 197, 199 when colliding superclusters are merged. The authoritative name and authoritative package are identified 197, 199 for each file, and the information resulting from the correct authoritative name can be associated with the file.

[00119] It should be noted that the steps are not necessarily performed strictly in the order listed above. In an implementation, it may be desirable to perform steps in parallel or in alternate orders. For example, the hashes of file contents can be collected while clustering the hashes of identical files, and even the similar file contents can be collected and even merged into clusters, in parallel to these steps. Indeed, it may be desirable to perform some of the steps in reverse order. However, the steps will be discussed as if they occur in order to avoid obscuring the inventive aspects.

[00120] FIG. 2 is a somewhat more detailed block diagram of the concept of using change frequency to identify the authoritative package, illustrating a supercluster 241 of files with identical and incrementally changed contents, broken down into package clusters 201, 221, 231.

All of the files in the supercluster 241 have identical contents, or have incrementally changed contents (as further explained below); here the files are represented by File X 2.0 (multiple copies), File X 2.1 (multiple copies), File X 2.1.2 (multiple copies), File X 2.2 (multiple copies), and File X 2.3. These files are preliminarily clustered in the same supercluster by the analysis because they are considered to be basically the same files as they have been changed from time-to-time and released in packages. The files in the supercluster 241 have been broken into package clusters 201, 221, 231; each package cluster has the same package clustered by version. Package cluster P 201 has all of the packages P 1.2, P 1.2.1, P 1.3, P 1.5, and P 1.6 203-211; package cluster Q 221 has all of the packages Q 2.2, Q 2.3, and Q 2.6 223-227; and package cluster R 231 has all of the packages R 3.0, R 3.1, R 4.1, and R 4.2 233-239. It can be seen that package cluster P 201 has 5 different package releases, and has 2 different versions of file X; package cluster Q 221 has 3 different package releases, and has 3 different versions of file X; and package cluster R 231 has 4 different package releases and has 3 different versions of file X. Each package release of package Q 221 includes a different version of file X. Packages P and R were each released several times without a new version of file X. Accordingly, the most change frequency for file X across versions is in package Q, which is identified as the authoritative package.

[00121] B. Details

[00122] 1. Identify the contents of every file in the ecosystem(s) -- Collect the hash of every Javascript file across the various ecosystems

[00123] The purpose of this step is to be able to identify the contents of every single file in the ecosystem. Recall that the higher level problem is to observe a file, such as a Javascript file, in a packaged application and tell the user what is known about the file. The first step is to recognize all the ecosystems that are supported (or are otherwise of interested), to recognize all of the files that are in the ecosystem, and then to identify which of those files have contents which are identical to each other.

[00124] To identify file contents, it is convenient and expeditious to hash the file contents. Hashes can be performed against file contents without regard to whether the file contents are

source or binary, for example. Note that identical hashes are not discarded. Note that source information indicating the source where the file originated is remembered, such as the package and/or ecosystem in which the file is located. Source information where the file originated may be included in a package in which the file was distributed.

[00125] The ecosystems which are to be included can be determined in various ways. In an implementation, there is an ongoing process of tracking the ecosystems that the system supports, for example, where software is coming from (components from existing repositories which the system then goes to and ingests). In another implementation, customers tell system where the software is from; some software is already known to the system. There may be an ongoing re-evaluation as to the ecosystems which are included, for example, the system can pull in a newly located pocket of Javascript and see how it fits into already-known information. The ecosystem can include the entire world of things are relevant. By using a large ecosystem, the unknowns can be decreased by increasing the number of knowns. The system can auto-discover repositories of its customers; techniques are known to discover the ecosystems. Most of the ecosystems include known sources with known structure, and files can be retrieved or observed using known techniques.

[00126] Accordingly, the ecosystem can include hundreds of thousands of components, and probably thousands of new ones every day. This is a Big Data problem.

[00127] In a variation, the ecosystem is limited to a specified universe of files.

[00128] 2. Determine all files which have identical contents -- Create file hash clusters (FHC) for all Javascript files, in which each cluster has only files with identical contents

[00129] The system tries to find all of the files that have identical contents. Conventionally, the file with the same contents has been distributed across thousands of packages. The system clusters together the files which have the same identical hash, which are known to exist at certain places.

[00130] Referring to a known Javascript ecosystem as an example, there are 42,000,000 files in that ecosystem (and increasing). However, accounting for uniqueness, there are only 6,000,000 unique files which have been distributed across packages. 36,000,000 files are just

duplicates of the same file throughout the entire Javascript ecosystem. The present system identifies the unique hash, and lists all of the places that unique hash (all of the identical files) exists. In this example, the system identifies 6,000,000 unique groups, each of which is a cluster of files with the same identical file contents.

[00131] Any way to identify files with the same contents may be used. Conventional hashing is convenient to use. The purpose is to identify unique files.

[00132] The system can collect all of those hashes, including information where the hash is from which is used later to answer the question regarding which one of these multiple places that the unique file came from is most likely to be the authoritative source of the unique file.

[00133] A file hash cluster, per unique file, identifies all of the places that the unique file was located in the ecosystem.

[00134] Note, in conventional identification using hashing of a file, identical files conventionally are discarded.

[00135] 3. Determine all files which have contents which are incrementally changed from the identical files -- Aggregate/enlarge file hash clusters on AST similarity

[00136] The purpose of this step is to incorporate files that should be in a file hash cluster (from the previous step) because they are semantically identical or incrementally different, but are not included because they have any change from the previous file.

[00137] Consider, for example, if the difference between files is even so small as a carriage return/line feed or white space, then the hash will not be identical even though the files are semantically identical.

[00138] A convenient way to identify semantically identical files is to prepare syntax abstraction of the structure of program code in the files to be compared, or more particularly, using an AST (abstract syntax tree) according to any conventional technique. An AST is known to represent the code structure in a way which ignores most non-syntax elements. An AST can be created of all of the files, and clusters which have files with identical ASTs can be combined. Furthermore, clusters which have files with sufficiently similar ASTs can be combined. An AST can be prepared from a binary file's machine code, or bytecode, or source code. ASTs

prepared from a source code and binary code version of the same file should be considered to be highly similar

[00139] The file input to the AST can be normalized prior to preparing the AST, to will alleviate minor differences in the files. Techniques are known for normalizing files.

[00140] A group of files which are identical, or with sufficiently similar contents, is referred to as a "supercluster".

[00141] The intention is for the system to catch files which have almost the same contents but for some reason the contents are not exactly the same.

[00142] As an example, consider that the file jquery will have incremental modification(s) across versions. The system intends to capture all of the jquery files with those incremental modifications into one cluster because we want to capture all alike files and files with small changes, to be grouped into the same clusters. As the file jquery changes from 1.1 to 1.2 to 1.3, there is going to be enough of a structural similarity in the file contents between 1.1 and 1.2, and between 1.2 and 1.3, that the files will match (most of the time) as part of this step, which will enlarge the clusters by including such incrementally changed files in the supercluster with the identical files.

[00143] The AST is a known technique which is convenient to use to identify semantically identical and semantically similar file contents because similarity can be identified using the tree structure. For example, files with identical ASTs of their file contents can be placed in the same superstructure. In addition, the system can set a percentage of similarity of ASTs so that files with highly similar ASTs can be placed in the same superstructure.

[00144] One way to aggregate and enlarge the file hash clusters is to use AST similarity, and variations. Thus, this step can include, for each file hash cluster merge (iterative; where similarity disperses across versions):

[00145] (i). Perform FHC AST similarity comparison and merge similar FHC

[00146] (ii) Perform minified AST/File inclusion and merge similar FHC

[00147] (iii) Perform normalized AST inclusion and merge similar FHC

[00148] Each of these is further considered below.

[00149] (i) Perform FHC AST similarity comparison and merge similar FHC

[00150] Consider an example of a first file hash cluster in which the members are identical (that is, they all have the same hash), such as JQuery 1.1.js. The system will look to the other file hash clusters to identify files which are, e.g., 95% or more similar to the first file hash cluster. This might identify a second file hash cluster with JQuery 1.2 as being 95% or more similar, and the system will incorporate the second file hash cluster into the first file hash cluster. And then the system will take the 1.2 version (the second file hash cluster) and search for similarity for that version, possibly identifying a third file hash cluster with JQuery 1.3 which is changed a little bit and is 98% similar to 1.2 and incorporate the third file hash cluster with JQuery 1.3 into the first file hash cluster. Similarity between the original file JQuery 1.1 and JQuery 1.3 version might be 80%.

[00151] The purpose is not to put consecutive versions together as such. Instead, the purpose is to find files whose contents are incrementally modified versions of 1.1. The system intends to identify 1.1, 1.1', and so on, and pull them into a supercluster. For example, assume that a user is given a canonical version and makes a slight change. And another user makes another slight change to the second version. The resulting third version may not match the canonical version but it matches the second version. The system brings the versions together into a supercluster so as to be able to create the version groups (in step # 4) by pulling apart the supercluster.

[00152] Here, the system uses the clusters of files which have identical contents, and then finds files which are incrementally slightly different (using, for example, an AST to determine source files which are slightly different) and add those incrementally modified files are incorporated into the supercluster. The supercluster thus will include identically same files and incrementally different files. For example, in a situation where there are clusters A, B and C, each containing identical files, cluster BB is incrementally different from A, and cluster C is incrementally different from B. Cluster B is added to cluster A since the files of B are incrementally different from A, and then cluster C may be added to cluster A since the files of C are incrementally different from B, which creates a supercluster of A+B+C. Note that the files

of C were not added originally to cluster A because C is too different from A and was only discovered via the inclusion of cluster B from which C is only incrementally different from A.

[00153] (ii) Perform minified AST/File inclusion and merge similar FHC

[00154] Minification is a technique used to shrink a total size of a file. Minification techniques are known to change a variable name which is too long into a smaller name; minification removes white space; some minification algorithms perform optimization of code. Many files are distributed in a minified form. For example, because Javascript is usually downloaded with the webpage, it is should be distributed in a very small form. When a file is delivered to a machine, the human readable version is usually minified to a very small size while achieving equivalent functionality. A regular Javascript file is not identical to the same file minified. Nevertheless, the system intends to match the minified files to the non-minified file, and pull the minified files into the supercluster.

[00155] In this system, some files in the ecosystem have already been minified. When a package is distributed, conventionally the non-minified and minified Javascript file are in the package, and the minified file is conventionally what is used in the application. So, the system has access via the ecosystem to both the minified and non-minified version of the file. When the ASTs are matched, the AST of the file probably will not match the minified version of the file.

[00156] One way to include the minified version of a file in the cluster that has a non-minified version is to search for and locate the minified version of the file based on the file name, and pull the minified version(s) into the supercluster which includes the non-minified version of the file. Note that the minification algorithm generates a well-defined minified file name pattern, e.g., with a .min extension. The end result is to pull the minified files into the appropriate supercluster.

[00157] (iii) Perform normalized AST inclusion and merge similar FHC

[00158] In an embodiment, the AST similarity and/or minification AST inclusion can be replaced by normalized AST inclusion.

[00159] Normalization can be utilized to find superclusters. The AST allows the system to ignore, e.g., white space and comments which can be ignored because they don't mean anything semantically. Other elements which are semantically irrelevant should be ignored before creating the hash.

[00160] In this variation, the AST is used to generate the hash of the file. This is another embodiment for determining what is included in the AST for generating the fingerprint.

[00161] The point is to find files which are similar so as to create superclusters by aggregating incrementally similar files (identified via minified, AST with/or without normalization).

[00162] 4. For a cluster of identical/incrementally changed files, determine the packages the files belong to, further break those packages down by versions, and identify the package with the "most change" frequency for that file -- Create package clusters from file hash clusters

[00163] Consider a situation that supercluster has been created pursuant to the previous steps #1 to #3, and in this example supercluster there are different javascript packages, e.g., a clipper package with jquery inside of clipper; a foo package with jquery, and another package is jquery with jquery inside the jquery package, and so on. This supercluster has all of the identical and incrementally similar files across all of the different packages (in the ecosystem) in which that file occurs. For a normal ecosystem, one supercluster may have hundreds of different packages in which the identical and incrementally similar jquery file occurs.

[00164] From that supercluster which has all of the different packages, the analysis will determine which of these hundreds of different packages which have jquery inside, is the *authoritative package*, in this example, for jquery. In other words, the analysis intends to determine out of all of these packages with jquery in it, which is the original project that is responsible for that file, e.g., jquery.

[00165] The analysis breaks the supercluster down into all of the packages therein. For each one of these packages, the supercluster breaks the package down into versions. The analysis will determine which package in which that file has most change across those versions, which is therefore considered to be the project where that file originates.

[00166] This analysis is based on the inventor's realization that one would expect when checking in a new jquery that it is changed, which is the reason to create a new version of that file. Consequently, breaking up the supercluster into the packages and then into versions and then looking for the files most changed across each version should identify the package (and thus the project) where the file originates.

[00167] In order to break the supercluster into the packages and versions, and then determine the "most change" frequency, the following steps are taken:

[00168] 4.a. Generate package names and versions for each member

[00169] 4.b. Aggregate on package name to determine version count

[00170] 4.c. Determine "most change" frequency across versions for each package by file Hash, and identify this as the authoritative package for that file.

[00171] Refer to the example of FIG. 3 which is representative of a jquery hash across multiple versions. Assume that the files which are identical and incrementally similar to jquery have all been pulled into the same supercluster. The analysis does not need know that these are all referred to as "jquery" and indeed similarity of name is not used in the analysis. More importantly, without further analysis it is unknown which of these packages is the original.

[00172] In a supercluster, the file names are probably not all the same, the hashes probably are not the same, and the versions will probably vary widely (as in the over-simplified example of Table 1). This overall analysis intends to determine which of these files is the authoritative file. In this step #4, the analysis looks in one supercluster for the package which has the most change across the versions; this is considered the authoritative package.

[00173] Reference is now made to the example oversimplified supercluster of Table 1. The third column is the "package name". A supercluster may include a thousand packages all of which have the identical and incrementally similar files in it. In this step #4, the analysis looks at the packages in one supercluster, puts all of the jquery packages into the same package cluster, and puts all of the Clipper packages into another package cluster, putting the webjar packages into yet another cluster, and so on, based on package name. Thus, the supercluster is broken

down into package clusters, and each of the package clusters has the packages with the same package name.

[00174] The hypothesis is that the author of jquery is pushing new packages that have the jquery file, in which the jquery file ideally changes with every released package. In other words, the original author of jquery is not going to release the package with the jquery file unless the jquery file is changed. However, if a developer is merely a user of some package with jquery, the developer is not going to change the version of jquery with every release of that package; the version of jquery may be changed in some releases but not every release. So the goal of step #4 is to determine, of the packages in the supercluster, which package has the file that changes with every release of that package. This is the authoritative file, because no one but the authoritative developer of that file will be creating a new version of the file with every release. This is what the "most change" is about.

[00175] In this step, all of the identical and incrementally similar files have been grouped into a supercluster, then the supercluster is sorted out into individual packages (each, a package cluster which has packages that have the same name), each package cluster is split into the version clusters (each version cluster has only the files with all the same version within the package cluster), and then from the frequency of change of version clusters within each package cluster, the analysis can determine which package cluster is the authoritative package.

[00176] "Change frequency" refers to how frequently the file in a package is changed, in relation to how frequently the package is released. The general concept is, in an ideal situation, every time a package is released the file is a new version; therefore this package must be the authoritative source. In reality, a file is not necessarily changed in a release of the authoritative package. Consequently, the package in which the file changes most frequently in relation to package releases is in the package which is the authoritative source. Identical files in a package cluster are considered to be only one "change" which is counted once; all changes to files in the package are counted. In determining change frequency, identical package releases in a package cluster are considered to be only one "change" which is counted once; all packages releases are counted.

[00177] "Change frequency" (sometimes referred to as "frequency of change") may be implemented as a ratio of change (such as number of versions of that file in a package vs. number of package releases of that package). The highest ratio of package releases to file changes is what is looked for. It is expected that this ratio of the package which is the authoritative source is close to 100 % in which each release (or almost each release) of that package has a change in that file. For consumers of the file, it is expected that the ratio is closer to 0-10%, e.g., for every 10 releases of a non-authoritative package, that package has a new jquery.

[00178] 4.c.i. Example of jquery hash across multiple versions

[00179] Reference is now made to FIG. 3, which is representative of a hash across multiple versions of the same or incrementally changed file jquery. As illustrated in the simplified file listing 301, under each hash is a file which has identical contents. Note that each file is listed so as to identify where in the ecosystem the file was obtained from. The files are named according to naming conventions.

[00180] 4.c.ii. Examples of frequency of change across versions

[00181] In this context, a change means a "change to the javascript file itself." When a project uses, a file such as jquery, the file jquery might be used as a dependency in which case there is no change from the original file; or it may be checked it into the source control and possibly edited (or a copyright line change, etc.) and any new character will make the hash different. This is considered a "change" to the file.

[00182] In the following example cases, the frequency of change is used to determine which package is the authoritative package. In all of the following cases, JQ should be determined to be the authoritative name of the authoritative package.

[00183] Consider the following cases a, b and c which compare frequency of change of the file jquery across versions of project JQ (which is the project in which jquery is developed) and project Foo (in which a developer merely uses jquery):

[00184] Case a. JQ: 10 versions, 10 changes & Foo: 12 versions, 10 changes

[00185] Case b. JQ: 10 versions, 10 changes & Foo: 10 version 10 changes

[00186] Case c. JQ: 10 versions, 8 changes & Foo: 10 version 10 changes

[00187] Case a is the normally expected case, in which every check-in is of a new version and in every check-in the file changes (in project JQ). There might be 12 versions of a corresponding project (here, project Foo), but jquery is not updated in every version of the corresponding project Foo; in this example jquery is updated in only two versions of the corresponding project Foo. A developer who merely uses jquery will release their project more frequently than the developer updates the file (e.g., jquery) which they are merely using.

[00188] In Case a, JQ is identified as the authoritative package.

[00189] In Case b, it just happens that the number of changes is equal in both the file and the project, so that both have the same highest frequency of change. In this situation, the system can use the file name to resolve which project is authoritative. Alternatively, the file can be forward for human review to resolve which project is authoritative.

[00190] Case c is a hypothetical case which is speculated must exist which causes the incorrect result. In Case c, there are more changes in what would be the incorrect project; causing the system to incorrectly identify the authoritative package. This might happen if, for example, in every check-in the developer of Foo adds their initials to the file, thereby creating a change for every single file in Foo. In this hypothetical scenario, the system will identify the wrong project (Foo) as being the original project. The system can resolve some of these errors by making sure that the ecosystem is correct, e.g., fee can default to the javascript ecosystem. Another possibility is that ecosystems might be assigned in pre-determined priority. For example, it is unlikely that a javascript developer would release via NuGet, and so a file of type javascript which identifies a nugget package as the authoritative project will be indicated as incorrect and needing further resolution.

[00191] In Case c, Foo is identified as being the authoritative project, which is known to be incorrect because it is in the wrong ecosystem for a javascript-type file. Because Foo is the wrong system, the result can be forwarded for human review.

[00192] In a variation, to resolve package inclusion, node modules should not participate in the frequency count.

[00193] 5. Pull any outlying files back into the package version group in which they belong -- Create package version groups from the package cluster based on authoritative package versions

[00194] Once we have identified the authoritative package, we need to move all of the other files into their correct version slot. From the supercluster and the frequency of change across versions, the analysis in the above Case A has authoritatively determined that JQ is the authoritative package. We have these other files that are in the supercluster but are not in the JQ package, such as listed in Table 1 and Table 2, namely, Clipper, Webjar, and NuGet. These package clusters are all in the same supercluster, but each package cluster should be assigned to a package version group, so as to authoritatively indicate that a (in the package cluster) file is most similar to an authoritatively named file, e.g., JQuery 1.1. Step #5 is intended to pull in the outlying authoritative packages in the version cluster into the authoritative package version group.

[00195] This can be done by:

[00196] a. Merging authoritative package outliers into authoritative package version groups; and

[00197] b. Merging minified outliers based on file name and not AST

[00198] As an example, consider the simplified example in Table 1, which has 6 packages all with JQ. There are six files in this simplified supercluster, and assume that JQ has been determined as the authoritative package. Now, doing an AST similarity comparison, the analysis finds that Clipper version 4.5 is most similar to Pkg 1 version 1.1, so it is assigned to the version 1.1 version cluster as shown in Table 2. When we see that clipper 4.5 inside an application, we can give it the data from the project being jquery 1.1. This step is about, how do the outliers get moved into the package version groups.

[00199] This happens in two ways. The non-minified files are optionally normalized and merged based on AST. The minified files can be merged based on file name and not on AST similarity.

[00200] **TABLE 1 - Supercluster of 6 package clusters all with file JQ**

	Package version	**Package name**
Pkg 1	1.1	JQ
Pkg 2	1.2	JQ
Pkg 3	1.3	JQ
Pkg N	4.5	Clipper
Pkg X	1.1.1	Webjar/JQ
Pkg Z	1.4	NuGet/JQ

[00201] **TABLE 2 - Versions Clusters Based on JQ Being Authoritative Package**

Authoritative Package	**Package Name and Version**	**Package Name and Version**	**Package Name and Version**
JQ 1.1	Pkg 1 1.1	Clipper 4.5	Pkg Y 1.4
JQ 1.2	Pkg 2 1.2		
JQ 1.3	Pkg 3 1.3	Pkg Z	

[00202] Collision resolution when assigning the outlying files may be handled. It is possible that the file did not change from 1.1 to 1.2, such as when a new release is created due to some other file in the package. If that is the case, then Clipper 4.5 might match both 1.1 and 1.2 in Table 2. The analysis can arbitrarily resolve the collision by assigning Clipper 4.5 to the earlier ("introduction") version. Accordingly, the analysis can resolve version collisions to introduction version.

[00203] 6. Resolve authoritative name collisions among the clusters of files -- Generate AName from authoritative package version groups

[00204] At this step, the analysis has all of the superclusters, and each supercluster has been assigned an authoritative name and an authoritative versions. Referring back to the examples of Tables 1 and 2, perhaps jquery versions 1.1 and 1.2 were in the same supercluster but jquery 1.3 was in another supercluster because it was a major rewrite and thus insufficiently similar to be identified as an incremental change. In that situation, the authoritative name for the supercluster which includes jquery 1.1 is going to be "jquery" and the authoritative name for the supercluster which includes jquery 1.3 is also going to be "jquery." There is an authoritative name collision, which should be resolved. Because of the way that we generate the authoritative package name itself, we know that if a collision occurs, they came from the same package structure.

[00205] This step includes the following:

[00206] a. Use the authoritative package to generate the AName name

[00207] b. Use the authoritative package versions to generate AName_versions

[00208] c. Each member of the authoritative package version groups receives AName_(name, version)

[00209] d. Collision Resolution

[00210] i. When the AName_name collides and the version does not collide

[00211] 1. Create a new version in the authoritative package versions

[00212] ii. When AName_name collides and the version collides

[00213] 1. Merge clusters

[00214] For example, if the AName is jquery for 1.1 and 1.2, and the AName is jquery for 1.3, the only way that those names could match is if the package format for 1.1, 1.2 and 1.3 is exactly the same. Accordingly, those two superclusters should be (and are) merged together. So in the case where we have a jquery 1.1 and 1.2 in one cluster and jquery 1.3 in another cluster, we need to merge those two superclusters to add the 1.3 version to the 1.1 and 1.2 versions.

[00215] In this example as explained above, jquery 1.3 was originally in a different supercluster. At the end of the day, we independently calculated that the ANames of two different superclusters are the same. This in effect realizes that 1.3 should have been in the same supercluster. The change between 1.2 and 1.3 was so significant that they didn't get merged into

the same superclusters. This step addresses the problem that a change may be so significant that the relationship between the files was not previously realized.

[00216] This step recognizes that the analysis might not make the correct superclusters at step 1. Step 2 has localization files or ancillary files that are not going to change from version to version to version, and they will end up in a supercluster with one file. In this step #6, this analysis finds that the authoritative name of one superclusters collides with the name from another supercluster. These superclusters will be merged into the appropriate authoritative name. The analysis can include the information from those files as well.

[00217] Consider that a particular package has file1 and file2, which are not similar at all and just perform different functions in the package. When the files are associated to an AName, we want the AName of both files to be the same as well as having the version resolution. By combining the ANames at the end, the analysis collapses file1 and file 2 under the same AName.

[00218] The point of creating the supercluster is to pull in version history as well as to pull in other modifications in other packages. The point of this last step is to merge all of the ancillary files inside that package back together under the AName. The local additional files are another example, which should be under the AName.

[00219] The version assignments happen before they are collapsed. So when collapsed, they already have the versions assigned.

[00220] In another embodiment, step 6 might occur before step 5.

[00221] In the worst case scenario the outlier files might not match the version and might not have a AName and might need human review.

[00222] C. Flow Diagram

[00223] Reference is now made to FIG. 4, an overall flow chart illustrating a procedure for authoritative name analysis. The flow chart of FIG. 4 encompasses the flow discussed above.

[00224] By way of review, the follow features were discussed above:

[00225] Step1, Step 2. Collect unique identifiers of files into clusters to create hash clusters, each hash cluster contains identical files (and indicates their sources). In the example of FIG. 4,

at step 401, the analysis creates file hash clusters based on identical hash matches. At step 403, the analysis generates an AST fingerprint for each hash cluster.

[00226] Step 3. Include, to create superclusters, of similar files so as to capture incremental changes over time. In the example of FIG. 4, at step 405, the analysis aggregates and enlarges file hash clusters based on AST similarity. At step 407, the analysis performs the previous operation until the outlier set is exhausted. At step 409, the analysis pulls in outlier files (for example, outlier js files) where the AST fingerprints and the files' ASTs are a sufficiently good match. At step 411, for each member of the outlier set, the analysis measures AST similarity between the candidate file's AST and all hash cluster ASTs. At step 413, the analysis performs the previous operation until the outlier set is exhausted.

[00227] Step 4. Break supercluster down into package clusters. Within each of the superclusters, discover which package cluster within a supercluster is the "most changed", which is the authoritative package. In the example of FIG. 4, at step 415, at this point we have a set of populated FHCs where the members are js files that are either identical or similar based on AST matches. At step 417, the analysis creates package clusters from the file hash clusters.

[00228] Step 5. Create package version groups from the authoritative packages and merge in outliers into packages. At step 419, for each version within an authoritative package group within a package cluster, the analysis evaluates AST similarity distances to other files that belong to low variance package groups within the same package cluster. Then, at step 421, the analysis identifies the authoritative package. At step 423, the analysis evaluates packages within each package cluster and identifies patterns where version variances within the same package are relatively higher than version variances of other packages. At step 425, the analysis assigns the version to each file that belongs to a lower version variant package, for example, propagate the version number from the most similar file in the high version variant package group.

[00229] Step 6. Generate authoritative name from authoritative package version groups and resolving collisions of authoritative names across the different superclusters (merging the superclusters which resolve into the same authoritative name). In the example of FIG. 4, at step 427, the analysis resolves A-name collisions.

[00230] As a result, the authoritative package version and source of the file is identified. Subsequently, post-identification processing can be performed using that information, for example, to associate that information based on the authoritative package version and source to other files that are resolved to the same authoritative package group. In the example of FIG. 4, at step 429, post-identification processing can assign the project URL, and/or license information and/or metadata information from the high version variant package group to all other members of the same package cluster. For example, in a package in which a developer merely uses a variation of jquery, after analysis that variation of jquery is associated with the license information and metadata information from the authoritative package version and source from the originator of jquery.

[00231] V. Additional Example Implementations

[00232] This section will discuss additional concrete examples of implementation, which are well suited to demonstrate various implementation details.

[00233] FIG. 5 illustrates a computer system implementation. The procedure of FIG. 6 to FIG. 8 conveniently may be implemented on the computer system of FIG. 5, or any another apparatus which is appropriately configured and arranged.

[00234]

[00235] Referring now to FIG. 5, a block diagram illustrating relevant portions of a computer system 501 will be discussed and described. A computer system 501 may include one or more controllers 503, a processor 505, an network interface 509 for communication such as with a network 507, a memory 511, a display 515 (optional), and/or a user input device such as a keyboard 517. Alternatively, or in addition to the keyboard 517, a user input device may comprise one or more of various known input devices, such as a pointing device, a keypad, a computer mouse, a touchpad, a touch screen, a trackball, and/or a keyboard. The display 515 is representative of a display that may present information to the user by way of a conventional liquid crystal display (LCD) or other visual display, and/or by way of a conventional audible device (e.g., a speaker) for playing out audible messages. Portions of the computer system 501

are well understood to those of skill in this area and have been omitted to avoid obscuring the discussion.

[00236] The processor 505 may comprise one or more microprocessors and/or one or more digital signal processors. The memory 511 may be coupled to the processor 505 and may comprise a read-only memory (ROM), a random-access memory (RAM), a programmable ROM (PROM), and/or an electrically erasable read-only memory (EEPROM). The memory 511 may include multiple memory locations for storing, among other things, an operating system, data and variables 533 for programs executed by the processor 505; such programs can include one or more of the following: to cluster 535 files in the ecosystem into superclusters based on identical file contents; determine 537 which files have contents which are similar yet incrementally changed, and merge their superclusters, to capture incremental changes over time to the same file in one supercluster; for each supercluster 539: break down into package clusters which cluster together files from the same package, determine which of the package clusters has the most change frequency across versions (as the authoritative package); resolve 541 authoritative name for files based on authoritative packages across superclusters, and generate the authoritative name; resolve 543 any authoritative name collision; use 545 a hash of file contents to determine files with identical contents; use 549 an AST of file contents to determine files with similar contents; move 551 a minified file into a cluster of the non-minified equivalent; and a database 553 for other information and/or instructions used by the processor 505. The computer programs may be stored, for example, in ROM or PROM and may direct the processor 505 in controlling the operation of the computer system 1001. Each of these functions is considered in more detail herein, to the extent that it is not detailed elsewhere in this document.

[00237] Responsive to manual signaling from the user input device represented by the keyboard 517, in accordance with instructions stored in memory 511, and/or automatically upon receipt of certain information via the network interface 509, the processor 505 may direct the execution of the stored programs.

[00238] The computer system 501 can access one or more ecosystems, here represented by ecosystem E 567 on which is stored one or more files, probably stored in connection with

packages, here represented by package P and package Q 569, 571. Although the files are illustrated as accessed over the network interface 509, the files 569, 571 may be remotely and/or locally accessible from the computer system 501, over a wired and/or wireless connection. The illustrated architectural environment for this runs into the cloud, represented by network 507. The files and packages 569, 571 do not need to be limited to a database or a software repository 567. The software repositories conventionally have well formatted descriptions of contents, and the computer system 501 herein can be adapted to software repositories in the cloud. Note that the computer system 501 can be distributed over multiple computers. Techniques are known for accessing files and packages and information regarding the packages and versions, which may be located in a software repository 567; packages also are commonly known to include package identifying information.

[00239] Much of the details of the programs have been discussed above, and the following is provided to expand understanding without necessarily repeating what is discussed above, and to possibly explore variations.

[00240] The processor 505 may be programmed to cluster 535 files in the ecosystem into superclusters based on identical file contents; determine which files have contents which are similar yet incrementally changed, and merge their superclusters, which is intended to capture incremental changes over time to the same file in one supercluster. The ecosystem can be preliminarily identified as being one or more of the following, by way of example: those ecosystems which the computer system is aware of, and/or those ecosystems which have been identified to the computer system, a pre-identified subset of one or more of the foregoing, and variations thereof. Note that the determination about whether the files are identical is based on file contents, possibly using conventional techniques. A simple method to determine which files have identical contents is to create hashes of file contents; files having identical hashes have identical contents. Any minor change to file contents, even a change which has no semantic significance (such as a comment), will result in a different hash. A simple method to determine which files have semantically identical contents uses AST, for which various techniques are available. Also, techniques are known for comparing AST trees to determine whether source

code is identical (for example, by omitting semantically insignificant characters from an AST) or similar (for example, by defining how much of the AST trees should be identical). The intention is to capture the idea that differences between a file which is released in a first version and a second version of that same file (in a package) are due to incremental changes, and furthermore that such files (in a package) which have been incrementally changed should be grouped together to determine whether the package in which the file was found is released as frequently as the file is released. In addition, techniques may be used to normalize file contents prior to preparing a hash or AST. Techniques will be known to those of skill in the art for determining whether file contents are semantically identical or semantically similar. In this instance, it is desirable for the computer system to retain identifying information regarding where the file is located in the ecosystem 567, for example, a package 569, 571 where the file was located. Thus, the system can include an indication of a source within the ecosystem for each of the files in the supercluster; the indication of the source within the ecosystem can be used later to retrieve additional information from the authoritative package (based on the indication of the source within the ecosystem of the file) regarding the authoritative file. Information regarding the superclusters, the files which are clustered together into superclusters, and the information used to determine whether the files are identical/similar (e.g., file hash and file AST), package clusters, and so on, can be stored, e.g., temporarily, e.g., in storage 553. As further discussed herein, note that some superclusters may be merged together (sometimes also referred to as being aggregated) for further processing, such as when the analysis determines that files in superclusters belong in the same supercluster. Note that there may be a very large amount of data which is processed, and accordingly, distributed processing across multiple processors 505 and/or storage in the cloud may be contemplated by some implementations.

[00241] The processor 505 may be programmed to perform the following, for each supercluster 539 determined to be identical or incrementally changed: break down into package clusters which cluster together files from the same package, determine which of the package clusters has the most change frequency across versions (as the authoritative package). This has been discussed in detail elsewhere. The intention is to determine all of the packages (each, a

package cluster) in which the file has been released; and to determine how many versions of the file exist within each of the packages. The breakdown can be based on the indication of source within the ecosystem of each file, or other identification of a package to which each file belongs. Then, the frequency of changing the version of the file vs. the frequency of changing the version of the package is determined. Ideally, in the authoritative package, a file's version changes with every version of the package. As discussed elsewhere, the analysis can be adapted to determine whether a file which is non-identical (and thus appears be a different version), is actually no more than adding comments, or inserting initials, to a semantically identical file, for example. In this sense, a "version" refers not to a canonical version of a file, but version which is a variation of the file and which might (or might not) have been a canonical version as released.

[00242] The processor 505 may be programmed to resolve 541 the authoritative name for files based on authoritative packages across superclusters, and generate the authoritative name. This has been discussed in detail above.

[00243] The processor 505 may be programmed to resolve 543 any authoritative name collision. This also is discussed in detail above

[00244] The processor 505 may be programmed to use 545 a hash of file contents to determine files with identical contents. Various techniques are known to prepare a hash of file contents. A comparison of such hashes of file contents may be made to determine whether files have identical contents.

[00245] The processor 505 may be programmed to use 549 an AST of file contents to determine files with similar contents. Techniques are conventionally available to prepare an AST of file contents. Techniques are known to determine whether AST trees of file contents are identical, and/or whether AST trees of file contents are sufficiently similar (for example, given a pre-determined threshold). Techniques are known to normalize file contents (for example, to remove comments, white content, and the like) prior to or subsequent to preparing an AST tree of file contents.

[00246] The processor 505 may be programmed to move 551 a minified file into a cluster of the non-minified equivalent file. Any conventional technique may be used to identify a minified

file and the non-minified equivalent, for example, association in a download package, or starting with a non-minified equivalent from which a minified file is prepared according to conventional techniques. A minified file and its non-minified equivalent file are intended to be treated as if they are the same version, and thus may be included in the same supercluster and package cluster.

[00247] The memory 511 also can include other miscellaneous information in a misc. database 1057, along with the usual temporary storage and other instructions for other programs not considered herein.

[00248] The computer system 501 can accommodate one or more disk drives or removable storage (not illustrated). Typically, these might be one or more of the following: a flash memory, a floppy disk drive, a hard disk drive, a CD ROM, a digital video disk, an optical disk, and/or a removable storage device such as a USB memory stick, variations and evolutions thereof. The number and type of drives and removable storage may vary, typically with different computer configurations. Disk drives may be options, and for space considerations, may be omitted from the computer system used in conjunction with the processes described herein. The computer may also include a CD ROM reader and CD recorder, which can be interconnected by a bus along with other peripheral devices supported by the bus structure and protocol (not illustrated). The bus can serves as the main information highway interconnecting other components of the computer, and can be connected via an interface to the computer. A disk controller (not illustrated) can interface disk drives to the system bus. These may be internal or external. In some embodiments, the computer system 501 can access a virtualized infrastructure commonly referred to as a "cloud" or "cloud storage." The processor 505, memory 511, a disk drive, removable storage medium, and/or cloud storage are referred to as “computer-readable storage media” and provide non-transitory storage of computer programs and data.

[00249] It should be understood that FIG. 5 is described in connection with logical groupings of functions or resources. One or more of these logical groupings may be omitted from one or more embodiments. For example, an embodiment may omit the resolution of authoritative name collision, or an embodiment may use a method other than hashing to determine identical file

contents, or an embodiment may use a method other than AST to determine similar file contents, and so on. Likewise, functions may be grouped differently, combined, or augmented without parting from the scope. Similarly the present description may describe various databases or collections of data and information. One or more groupings of the data or information may be omitted, distributed, combined, or augmented, or provided locally and/or remotely and/or in a virtualized infrastructure commonly referred to as a "cloud" without departing from the scope.

[00250] FIG. 6 to FIG. 8 collectively are a flow chart illustrating a procedure to provide an authoritative name analysis. The procedure can conveniently be implemented on a computer system, such as illustrated in FIG. 5, or other computer system appropriately arranged.

[00251] The procedure for authoritative name analysis 601 will determine 603, for all files in the ecosystem(s), which of the files have identical file contents, and will cluster identical files into superclusters. It is anticipated that the contents of such files are components, for example, including source code. Accordingly, the procedure 601 has prepared superclusters so that the files in each supercluster of the plurality of superclusters have identical contents.

[00252] The procedure 601 will get 605 get the first supercluster or the next supercluster. For this supercluster 607, the procedure will determine 609 which of the other superclusters have files which, in comparison to the file in the present supercluster, have file contents which is incrementally changed; and if one of the other superclusters has files with files contetns that is incrementally changed in comparison to the files in the present supercluster, both the present supercluster and the one of the other superclusters will be merged, so that there is one supercluster which includes the identical files and the files which were incrementally changed. The procedure repeats 611 until all of the superclusters have been handled. There may be fewer superclusters after performing this step 609. Accordingly, the procedure 601 has prepared superclusters so that the files in each supercluster of the plurality of superclusters have identical and incrementally changed filed contents. Consequently, the procedure has determined, of the files in the ecosystem which are clustered into the plurality of superclusters, which of the files have similar contents to each other, and merged the files which have similar contents to each

other into the same supercluster, to capture possibly incremental changes to the files over time in one of the superclusters which has the files with identical contents and similar contents.

[00253] FIG. 6 then proceeds 613 to FIG. 7 at 701.

[00254] Then the procedure 601 will break each supercluster down into package clusters, and determine which of the package clusters has the most change frequency across versions, as the authoritative package. In more detail, the procedure 601 gets 703 the first (or next) supercluster. For this supercluster 705, the following are performed. The procedure breaks 707 the supercluster down by packages, where files were released in a package. Accordingly, each supercluster is broken down into package clusters, based on packages to which the files belong, each of the package clusters has the files from a same package. Then, the procedure determines 709 the frequency of change for each of the package clusters, and identifies the package with the highest change frequency which the procedure considers to be the authoritative package of this superclusters. Frequency of change is described herein elsewhere in more detail. This supercluster now has an authoritative name and version for its file. This is repeated 713 so long as there is another supercluster to handle. Accordingly, for each of the superclusters, the procedure has determined which of the package clusters has the most change frequency across versions as the authoritative package.

[00255] FIG. 7 then proceeds 715 to FIG. 8 at 801.

[00256] Then the procedure 601 will get 803 the first (or next) supercluster. For this supercluster 805, the procedure 601 will determine 807 whether there is any authoritative name collision, for example, where a version of a file was so substantial than it was not included in a same supercluster with other versions of the same file, and will collapse 809 authoritative name collisions to merge ancillary files inside a package back together under the same authoritative name. The procedure 601 will handle all of the superclusters 811 in the same way. At the end of this sequence, there may be fewer superclusters than at the beginning of this sequence, due to authoritative name collisions. Consequently, the procedure 601 has resolve an authoritative name for the files, based on the authoritative packages that are determined, across the plurality of

superclusters which have files with identical and similar contents, and generated the authoritative name; and has resolved any authoritative name collisions.

[00257] In operation of such an authoritative name analysis, the authoritative name and authoritative package of the file have been determined with much greater accuracy than conventional systems. That is, the original name of a file which appears in different packages is correctly identified, in combination with the original package which originated that component. Accordingly, visibility and understanding of packages and the software components has been greatly improved, despite the possibility that the ecosystem has numerous files which are essentially the same but with minor variations in content and/or name, and despite the likelihood that many of these components might have originated elsewhere, possibly not even in the ecosystem. Use of such correct information in post-identified processing will be appreciated, such as assigning license information and/or metadata information to all members of the same package cluster.

[00258] VI. Glossary

[00259] Terms as used herein are intended to be interpreted first, as understood to one of skill in the art at a first level, of software repositories; and if not interpretable at the first level, than at a second level as understood by one skilled in the science of tools that support and manage software development; and if not interpretable at the first level and second level, then at a third level as understood to one of skill in the art of computer science; and then if not interpretable according to the first, second and third levels, according to a general dictionary.

[00260] The claims may use the following terms (with or without capitalization), which are defined to have the following meanings for the purpose of the claims herein. Other definitions may be specified in this document.

[00261] The designation "File Hash Cluster" (sometimes referred to as FHC), when used in a claim, is defined as a collection of plural files where the contents of all of the files, as determined from hashes of the contents of the files, are identical. Techniques are available for creating a hash, including a hash of contents of a file.

[00262] The designation "Abstract Syntax Tree" (sometimes referred to as AST) is defined as a representation of the abstract syntactic structure of source code written in a programming language, wherein the nodes of the tree each denote a construct occurring in the source code. Techniques are available for creating an abstract syntax tree from source code.

[00263] The designation "Package Cluster" is defined herein as a collection of files, all of which are from the same file hash cluster (and therefore have identical contents), and file hash clusters which have files which are determined to be incrementally different from those in the same file hash cluster, in which the collection of files in the package cluster all have the same package name.

[00264] The designation "package" as used herein is defined as a container (which is a file or a folder) that bundles together components intended for use by a particular application, and/or to assist in installing and/or de-installing, and/or running the particular application, and/or which are used by the particular software application and/or which are used in a build of the particular software application; the package also includes manifest data which identify the package and which uniquely identify the constituent files that enable the package to function. By way of explanation and not definition, the package may contain, as a non-limiting example, one or more of the following: content needed to reference an external library, source code, binaries, executable files, classes, interfaces, scripts, documentation, components used during installation, components used for running the particular application itself, other packages, components and/or related items. The manifest metadata in a package generally indicates the following or similar: the software's name, description of its purpose, version number, vendor, integrity information such as a checksum, and an enumerated list of dependencies on other components necessary for the software to run properly. Package managers which consume packages include, by way of example, package managers offered under the trade-marks and/or trade names NuGet, Node Package Manager, PyPi, RubyGems, PHP Composer, Android, Alpine Linux, Windows 8 and later, Windows Phone, Debian and its derivatives, Gentoo Linux, ORB Launcher, Pardus, OSDX, iOX, PlaysStation 3, Solaris, SunOS, UNIX System V, Symbian, BeOS, Apple Netwon, Puppy Linux, Red Hat Enterprise Linux, Fedora and its derivatives, Ubuntu, Slitaz, Tiny Core Linux, Ach Linux's Pacman, Slackware, and evolutions and extensions thereof. Formats of packages have internal structures and file name extensions consistent with one or more package

formats defined under the trade-marks or trade names of, for example, APK, APPX, APPXBundle, Deb, ebuild, ORB, PISI, PKG, PUP, PET, RPM, SNAP, TAZPKG, TCZ, Tar archives, and evolutions and extensions thereof.

[00265] The designation "Package Version Cluster" is defined herein as a collection of files, all of which are from the same package, in which the collection of files all have the same version number.

[00266] The designation "Authoritative Package" as used herein is defined as the package, in a package version cluster, that has the most change across versions.

[00267] The designation "Authoritative Name" as used herein refers to the name and version that represents the project responsible for the package creation and maintenance of the authoritative package.

[00268] The designation "Collision Resolution" refers to an algorithmic choice to determine direction when two items resolve to the same thing due to similar data.

[00269] The designation "minification" or "minifying" refers to a process of removing unnecessary characters from source code without changing the functionality of the source code; the unnecessary characters may include, by way of example, one or more of white space characters, new line characters, comments, and/or block delimiters, and similar characters, all of which are not required for the code to execute, but which typically are intended to add readability to the code.

[00270] The designation "minified" refers to code which resulted from minification of source code.

[00271] By way of explanation and not definition, an AST of minified source code typically is different from an AST of the same source code which is not minified.

[00272] The designation "normalized AST" or "normalized Abstract Syntax Tree" refers to an AST in which the name and/or structure variants within the source code have been removed or replaced to allow comparison of functionally equivalent source code. Techniques are known to

create a normalized AST, which can be created from code which has been normalized, and/or by normalizing the AST itself.

[00273] The term "byte code" as used herein is defined to mean an intermediate operational code which was generated by translating an input programming language, and then the byte code can be interpreted at the time the byte code is executed by the virtual machine which has its own instruction set independent of the native machine code for the hardware on which the virtual machine executes. Examples of computer languages which use "byte code" include, without limitation, Java, .NET, Scala, jython, groovy, and Pascal

[00274] The term "component" used herein is defined to be a specific version (which may be a canonical version) of pre-existing software, or a reusable pre-existing self-contained software code building block which is not a complete stand-alone finished product ready for use, which is source code, software or run-time executable code; a component can comprise one or more nested components within the component itself; a component can be the subject of a risk such as a license or a target of a security vulnerability.

[00275] By way of explanation and not as a definition, a component which is part of a stand-alone product can be understood to be a self-contained bit of code which a developer does not wish to write himself/herself as part of the stand-alone product, and so the developer uses a previously existing component for which the functionality likely was previously vetted as a part of another stand-alone product. As a non-limiting example of components which are nested, a Java web application packaged as a WAR component might contain various JAR components and JavaScript libraries, each of which is a component themselves.

[00276] The term "computer system" or "computer" used herein denotes a device sometimes referred to as a computer, laptop, personal computer, tablet computer, handheld computer, smart phone, personal digital assistant, notebook computer, personal assignment pad, server, client, mainframe computer, minicomputer, or evolutions and equivalents thereof.

[00277] "Open source" software is defined herein to be source code that allows distribution as source code as well as compiled form and/or binary form, with a well-publicized and indexed

means of obtaining the source, optionally with a license that allows modifications and derived works.

[00278] The term "repository" or "software repository" used herein are defined to mean an electronic storage system that stores components (sometimes referred to as an "artifact") and dependencies for later retrieval, to which artifacts are published in accordance with procedures which are well known to those in the field, so that artifacts made by one software developer are published to be available for further use by other software developers, to be incorporated as building blocks to build software products which can be executed; a repository can include a computer server by which the stored electronic copies of artifacts are made available for use by software developers to be incorporated as building blocks to build software products which can be executed; the repository typically has a unique identifier that indicates the software developer (an individual or a group) that contributed the artifact.

[00279] By way of explanation and not definition, a repository can be remote or local; some embodiments of a repository might be stored in a virtualized infrastructure commonly referred to as a "cloud." Examples of conventional software repositories include, by way of example but without limitation: the Central Repository (also known as Maven Central), the NuGet Gallery, RubyGems.org, npmjs.org, and many others. Repositories tend to rely on pre-defined formats and tools, for example, the Maven repository format, REST API interactions, different directory structures with format specific files for metadata, and the like. Software repositories are accessed by tools, including by way of example but without limitation: build tools such as Maven, Gradle, rake, grunt, and others; package managers such as npm, nugget, gem, and others; Integrated Development Environments such as Eclipse, IntelliJ, and many others.

[00280] The term "software build" used herein specifically is defined to mean a process as pre-defined in an executable build program of converting a plurality of components (some or all of which may be obtained from a repository) and combining the results into an executable stand-alone computer program or a software component for use in a further software build, including at least compiling components and linking compiled components and possibly binary components (which can be from the repository), in a pre-determined order as defined in the build program.

[00281] The term "compiler" is used herein specifically to mean a computer program(s) that transforms source code written in a programming language into a target language that is readable by a computer, often in the form of binary code or bytecode, so as to create an executable program.

[00282] The phrase "automatically without manual intervention," when used in a claim, is defined to mean that the particular step occurs after the step is initiated until limitations recited in the step are finished without requiring a user to provide input to a processor.

[00283] VII. Implementation and Technical Notes

[00284] The above discussion has assumed that the reader has a sufficient technical background for appreciating the points made. This section provides some supplementary implementation and/or technical notes which discuss some technical information that may be relevant.

[00285] The instant disclosure is provided to further explain in an enabling fashion the best modes of performing one or more embodiments. The disclosure is further offered to enhance an understanding and appreciation for the inventive principles and advantages thereof, rather than to limit in any manner the invention. The invention is defined solely by the appended claims including any amendments made during the pendency of this application and all equivalents of those claims as issued.

[00286] It is further understood that the use of relational terms such as first and second, and the like, if any, are used solely to distinguish one from another entity, item, or action without necessarily requiring or implying any actual such relationship or order between such entities, items or actions. It is noted that some embodiments may include a plurality of processes or steps, which can be performed in any order, unless expressly and necessarily limited to a particular order; i.e., processes or steps that are not so limited may be performed in any order.

[00287] Much of the inventive functionality and many of the inventive principles when implemented, are best supported with or in software or integrated circuits (ICs), such as a digital signal processor and software therefore, and/or application specific ICs. It is expected that one of ordinary skill, notwithstanding possibly significant effort and many design choices motivated

by, for example, available time, current technology, and economic considerations, when guided by the concepts and principles disclosed herein will be readily capable of generating such software instructions or ICs with minimal experimentation. Therefore, in the interest of brevity and minimization of any risk of obscuring principles and concepts, further discussion of such software and ICs, if any, will be limited to the essentials with respect to the principles and concepts used by the exemplary embodiments.

[00288] The various embodiments which demonstrate a method and/or system for providing an authoritative name analysis have been discussed in detail above. It should be further noted that the above-described processes can be stored as instructions in computer-readable storage medium. When the instructions are executed by a computer, for example after being loaded from a computer-readable storage medium, the process(es) are performed. The detailed descriptions, which appear herein, may be presented in terms of program procedures executed on a computer or a network of computers. These procedural descriptions and representations herein are the means used by those skilled in the art to most effectively convey the substance of their work to others skilled in the art.

[00289] Further, an embodiment has been discussed in certain examples as if it is used by a single developer or administrator at a single site. An embodiment may be used by numerous developers, administrators and/or related users, if preferred, at one or more sites.

[00290] A procedure is generally conceived to be a self-consistent sequence of steps leading to a desired result. These steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored on non-transitory computer-readable media, transferred, combined, compared and otherwise manipulated. It proves convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. It should be noted, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

[00291] Further, the manipulations performed are often referred to in terms such as adding, determining, or comparing, which are commonly associated with mental operations performed by a human operator. While the discussion herein may contemplate the use of an operator, a human operator is not necessary, or desirable in most cases, to perform the actual functions described herein; the operations are machine operations.

[00292] Various computers or computer systems may be programmed with programs written in accordance with the teachings herein, or it may prove more convenient to construct a more specialized apparatus to perform the required method steps. The required structure for a variety of these machines will be apparent from the description given herein.

[00293] A computer-readable storage medium is tangible and non-transitory; a computer-readable storage medium can be any of the memory or storage devices, such as those examples described above, or other removable or fixed storage medium, provided such computer-readable storage medium is tangible and non-transitory.

[00294] Furthermore, any communication network implicated in an embodiment can include, by way of example but not limitation, data and/or packet communications networks, which can provide wireless communications capability and/or utilize wireline connections such as cable and/or a connector, or similar. Any appropriate communication protocol may be used.

[00295] The computer and/or system embodied in connection herewith may (or may not) rely on the integration of various elements including, as appropriate and/or if desired, by way of example but not limitation, hardware and software servers, applications software, database engines, server area networks, conventional firewall and SSL security, production back-up systems, and/or applications interface software. An embodiment may be, by way of example but not by way of limitation, network-based and may (or may not) utilize a network such as the Internet or other network as an exemplary interface with the user for any information delivery.

[00296] One or more databases implicated by the above discussion may be, by way of example but not limitation, in a relational database format, but other standard data formats may also be used. Optionally, the various databases may include a known conversion system capable of receiving data in various standard formats.

[00297] One or more displays for the system may be developed in connection with, by way of example but not limitation, HTML display format, using XML. Although HTML, and XML, may be a preferred display format, it is possible to utilize alternative display formats for interacting with a user and obtaining user instructions.

[00298] This disclosure is intended to explain how to fashion and use various embodiments in accordance with the invention rather than to limit the true, intended, and fair scope and spirit thereof. The invention is defined solely by the appended claims, as they may be amended during the pendency of this application for patent, and all equivalents thereof. The foregoing description is not intended to be exhaustive or to limit the invention to the precise form disclosed. Modifications or variations are possible in light of the above teachings. The embodiment(s) was chosen and described to provide the best illustration of the principles of the invention and its practical application, and to enable one of ordinary skill in the art to utilize the invention in various embodiments and with various modifications as are suited to the particular use contemplated. All such modifications and variations are within the scope of the invention as determined by the appended claims, as may be amended during the pendency of this application for patent, and all equivalents thereof, when interpreted in accordance with the breadth to which they are fairly, legally, and equitably entitled.

WHAT IS CLAIMED IS:

1. A method for providing an authoritative name source for files within an ecosystem, comprising the following performed by at least one processor:

clustering the files in the ecosystem into a plurality of superclusters, in which the files in each supercluster of the plurality of superclusters have identical contents; then

determining, of the files in the ecosystem which are clustered into the plurality of superclusters, which of the files have similar contents to each other, and merging the files which have similar contents to each other into the same supercluster, to capture possibly incremental changes to the files over time in one of the superclusters which has the files with identical contents and similar contents;

for each supercluster which has the files with identical and similar contents:

breaking the supercluster down into package clusters, based on packages to which the files belong, each of the package clusters has the files from a same package; and

determining which of the package clusters has most change frequency across versions of the files within the same package, as the authoritative package, wherein change frequency refers to how frequently the version is changed in relation to how frequently the package is released; then

resolving an authoritative name for the files, based on the authoritative packages that are determined, across the plurality of superclusters which have files with identical and similar contents, and generating the authoritative name; and

resolving any authoritative name collision.

2. The method of claim 1, wherein

a hash of file contents of each of the plurality of files is used to determine which of the files have identical contents,

the clustering of the files in the superclusters with identical contents is determined based on the hashes of file contents,

for the supercluster, the supercluster includes an indication of a source within the ecosystem for each file of the files,

the breaking down into the package clusters is based on the indication of the source of the each file.

3. The method of claim 1, wherein

an abstract syntax tree of file contents of each of the plurality of files is used to determine whether contents of files are similar,

the determining of files with similar contents into the same supercluster is based on the abstract syntax tree of file contents,

for the supercluster, the supercluster includes an indication of a source within the ecosystem for each file of the files.

4. The method of claim 1, wherein breaking each supercluster down into package clusters includes

generating package names and versions for members of the supercluster,

aggregating package clusters on package name and counting unique changes to the package cluster, and

determining, for each package cluster, the most change frequency among the package clusters.

5. The method of claim 1, wherein

the authoritative name is resolved by

merging authoritative package outliers into groups which are merged into package cluster based on file name,

the resolved authoritative name and source are identified.

6. The method of claim 1, further comprising

determining whether the authoritative name source is correct based on ecosystem in which authoritative name source originated.

7. The method of claim 1, wherein the ecosystem, in which the files are located which are clustered, includes at least one software repository in which the files are located.

8. A non-transitory computer readable medium comprising instructions for execution by a computer, the instructions including a computer-implemented method for providing an authoritative name source for files within an ecosystem, the instructions for implementing:

clustering the files in the ecosystem into a plurality of superclusters, in which the files in each supercluster of the plurality of superclusters have identical contents; then

determining, of the files in the ecosystem which are clustered into the plurality of superclusters, which of the files have similar contents to each other, and merging the files which have similar contents to each other into the same supercluster, to capture possibly incremental changes to the files over time in one of the superclusters which has the files with identical contents and similar contents;

for each supercluster which has the files with identical and similar contents:

breaking the supercluster down into package clusters, based on packages to which the files belong, each of the package clusters has the files from a same package; and

determining which of the package clusters has most change frequency across versions of the files within the same package, as the authoritative package, wherein change frequency refers to how frequently the version is changed in relation to how frequently the package is released; then

resolving an authoritative name for the files, based on the authoritative packages that are determined, across the plurality of superclusters which have files with identical and similar contents, and generating the authoritative name; and

resolving any authoritative name collision.

9. The non-transitory computer-readable medium of claim 8, further comprising instructions for implementing:

a hash of file contents of each of the plurality of files is used to determine which of the files have identical contents,

the clustering of the files in the superclusters with identical contents is determined based on the hashes of file contents,

for the supercluster, the supercluster includes an indication of a source within the ecosystem for each file of the files,

the breaking down into the package clusters is based on the indication of the source of the each file.

10. The non-transitory computer-readable medium of claim 8, wherein

an abstract syntax tree of file contents of each of the plurality of files is used to determine whether contents of files are similar,

the determining of files with similar contents into the same supercluster is based on the abstract syntax tree of file contents,

for the supercluster, the supercluster includes an indication of a source within the ecosystem for each file of the files.

11. The non-transitory computer-readable medium of claim 8, wherein

breaking each supercluster down into package clusters includes instructions for implementing:

generating package names and versions for members of the supercluster,

aggregating package clusters on package name and counting unique changes to the package cluster, and

determining, for each package cluster, the most change frequency among the package clusters.

12. The non-transitory computer-readable medium of claim 8, wherein

the authoritative name is resolved by instructions for implementing:

merging authoritative package outliers into groups which are merged into package cluster based on file name,

identifying the resolved authoritative name and source.

13. The non-transitory computer-readable medium of claim 8, further comprising instructions for implementing:

determining whether the authoritative name source is correct based on ecosystem in which authoritative name source originated.

14. The non-transitory computer-readable medium of claim 8, wherein the ecosystem, in which the files are located which are clustered, includes at least one software repository in which the files are located.

15. A computer system that provides an authoritative name source for files within an ecosystem, comprising:

at least one processor, the at least one processor is configured to:

identify the ecosystem;

cluster the files in the ecosystem into a plurality of superclusters, in which the files in each supercluster of the plurality of superclusters have identical contents; then

determine, of the files in the ecosystem which are clustered into the plurality of superclusters, which of the files have similar contents to each other, and merge the files which have similar contents to each other into the same supercluster, to capture possibly incremental changes to the files over time in one of the superclusters which has the files with identical contents and similar contents;

for each supercluster which has the files with identical and similar contents:

break the supercluster down into package clusters, based on packages to which the files belong, each of the package clusters has the files from a same package; and

determine which of the package clusters has most change frequency across versions of the files within the same package, as the authoritative package, wherein change frequency refers to how frequently the version is changed in relation to how frequently the package is released; then

resolve an authoritative name for the files, based on the authoritative packages that are determined, across the plurality of superclusters which have files with identical and similar contents, and generate the authoritative name; and

resolve any authoritative name collision.

16. The computer system of claim 15, wherein

a hash of file contents of each of the plurality of files is used to determine which of the files have identical contents,

the clustering of the files in the superclusters with identical contents is determined based on the hashes of file contents,

for the supercluster, the supercluster includes an indication of a source within the ecosystem for each file of the files,

the processor is further configured to

break down the supercluster into the package clusters based on the indication of the source of the each file.

17. The computer system of claim 15, wherein

an abstract syntax tree of file contents of each of the plurality of files is used to determine whether contents of files are similar,

the determining of files with similar contents into the same supercluster is based on the abstract syntax tree of file contents,

for the supercluster, the supercluster includes an indication of a source within the ecosystem for each file of the files.

18. The computer system of claim 15, wherein the processor is further configured to break down the supercluster into the package clusters, to:

generate package names and versions for members of the supercluster,

aggregate package clusters on package name and counting unique changes to the package cluster, and

determine, for each package cluster, the most change frequency among the package clusters.

19. The computer system of claim 15, wherein the processor is further configured to

resolve the authoritative name, to:

merge authoritative package outliers into groups which are merged into package cluster based on file name,

identify the resolved authoritative name and source.

20. The computer system of claim 15, wherein the processor is further configured to

determine whether the authoritative name source is correct based on ecosystem in which authoritative name source originated.

FIG. 1

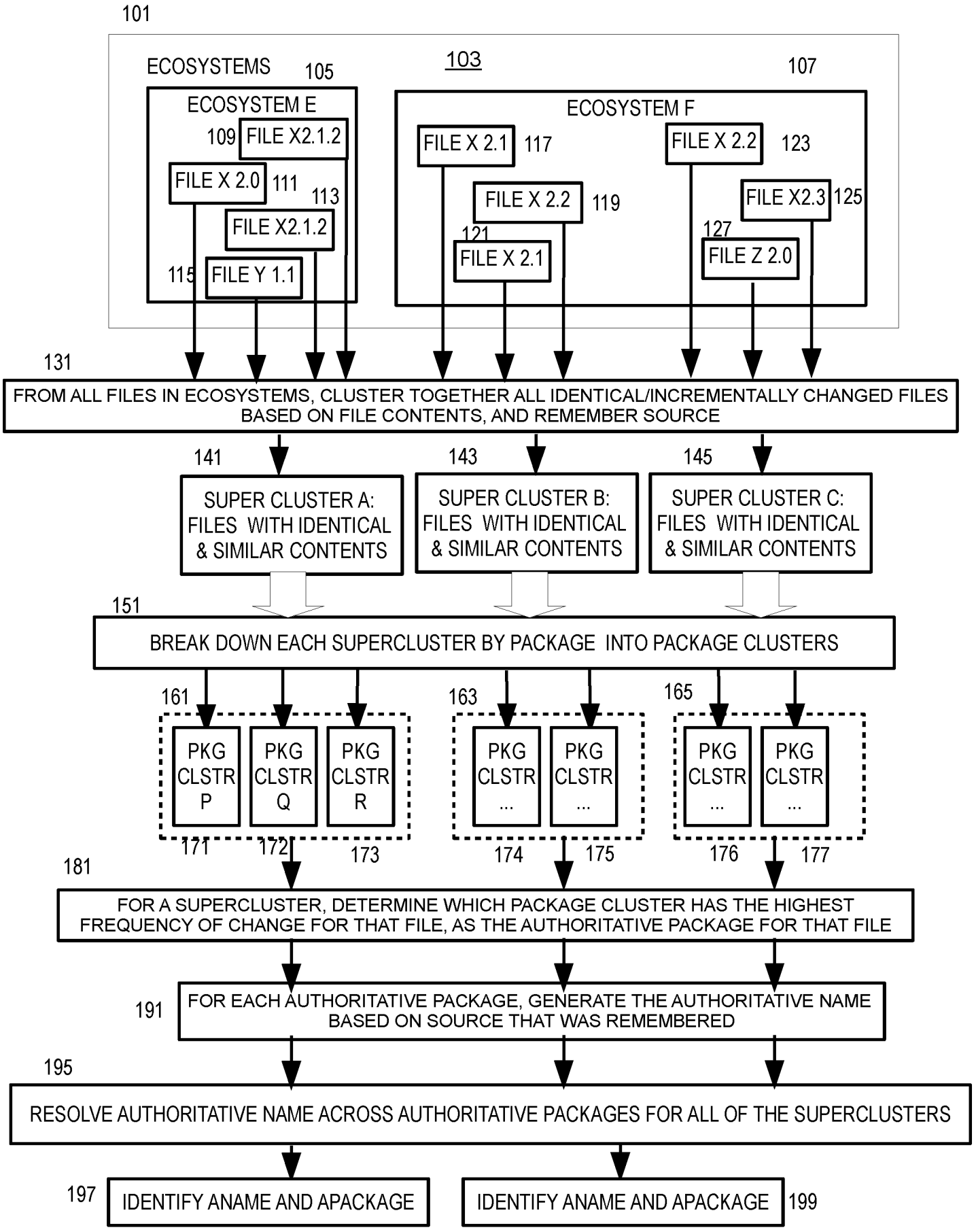

101
ECOSYSTEMS
105
103
107
ECOSYSTEM E
109
FILE X2.1.2
FILE X 2.0
111
113
FILE X2.1.2
115
FILE Y 1.1
ECOSYSTEM F
FILE X 2.1
117
FILE X 2.2
123
FILE X 2.2
119
FILE X2.3
125
121
FILE X 2.1
127
FILE Z 2.0
131
FROM ALL FILES IN ECOSYSTEMS, CLUSTER TOGETHER ALL IDENTICAL/INCREMENTALLY CHANGED FILES BASED ON FILE CONTENTS, AND REMEMBER SOURCE
141
143
145
SUPER CLUSTER A: FILES WITH IDENTICAL & SIMILAR CONTENTS
SUPER CLUSTER B: FILES WITH IDENTICAL & SIMILAR CONTENTS
SUPER CLUSTER C: FILES WITH IDENTICAL & SIMILAR CONTENTS
151
BREAK DOWN EACH SUPERCLUSTER BY PACKAGE INTO PACKAGE CLUSTERS
161
163
165
PKG CLSTR P
PKG CLSTR Q
PKG CLSTR R
PKG CLSTR ...
PKG CLSTR ...
PKG CLSTR ...
PKG CLSTR ...
171
172
173
174
175
176
177
181
FOR A SUPERCLUSTER, DETERMINE WHICH PACKAGE CLUSTER HAS THE HIGHEST FREQUENCY OF CHANGE FOR THAT FILE, AS THE AUTHORITATIVE PACKAGE FOR THAT FILE
191
FOR EACH AUTHORITATIVE PACKAGE, GENERATE THE AUTHORITATIVE NAME BASED ON SOURCE THAT WAS REMEMBERED
195
RESOLVE AUTHORITATIVE NAME ACROSS AUTHORITATIVE PACKAGES FOR ALL OF THE SUPERCLUSTERS
197
IDENTIFY ANAME AND APACKAGE
IDENTIFY ANAME AND APACKAGE
199

FIG. 2

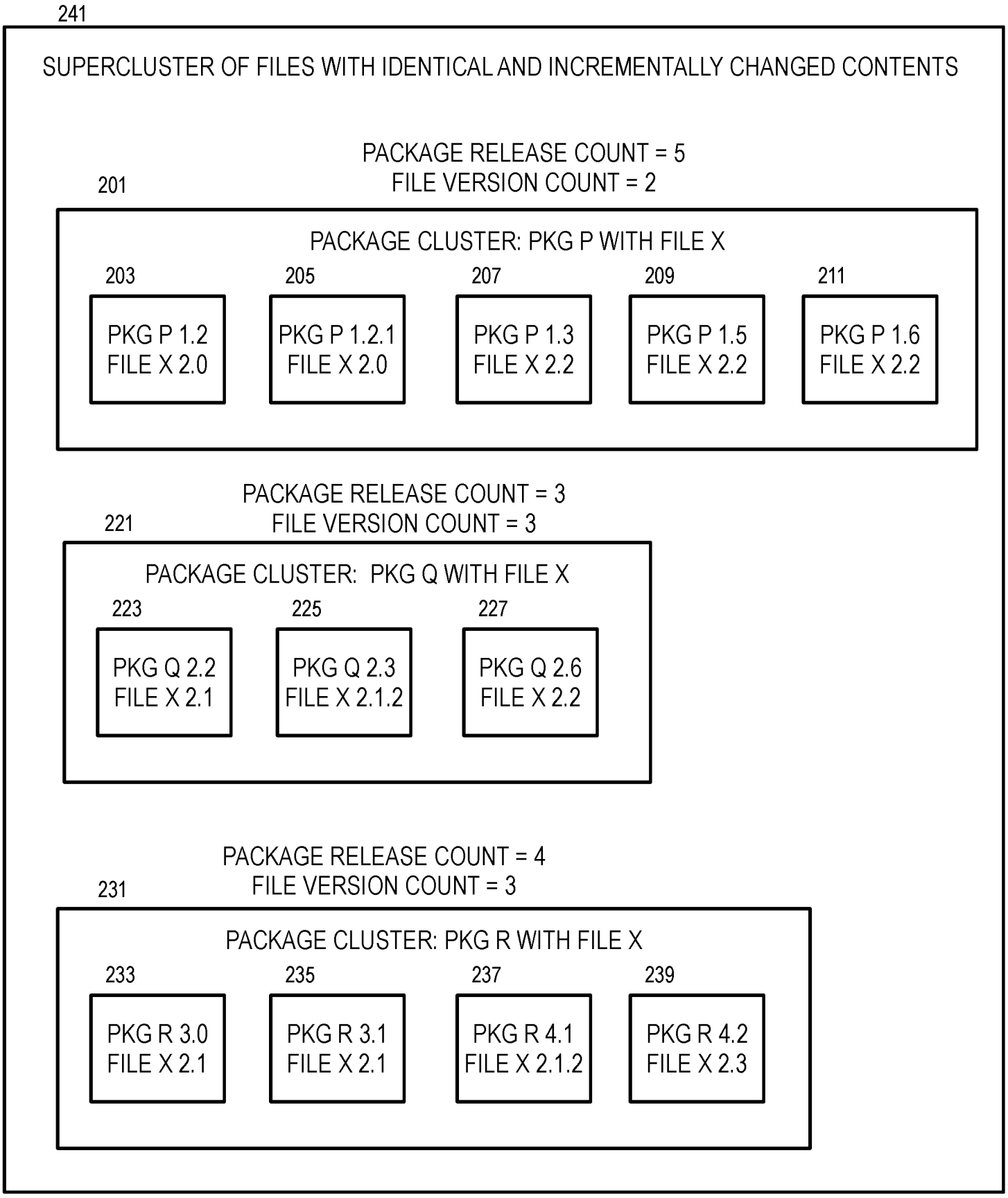

241
SUPERCLUSTER OF FILES WITH IDENTICAL AND INCREMENTALLY CHANGED CONTENTS
PACKAGE RELEASE COUNT = 5
FILE VERSION COUNT = 2
201
PACKAGE CLUSTER: PKG P WITH FILE X
203
PKG P 1.2
FILE X 2.0
205
PKG P 1.2.1
FILE X 2.0
207
PKG P 1.3
FILE X 2.2
209
PKG P 1.5
FILE X 2.2
211
PKG P 1.6
FILE X 2.2
PACKAGE RELEASE COUNT = 3
FILE VERSION COUNT = 3
221
PACKAGE CLUSTER: PKG Q WITH FILE X
223
PKG Q 2.2
FILE X 2.1
225
PKG Q 2.3
FILE X 2.1.2
227
PKG Q 2.6
FILE X 2.2
PACKAGE RELEASE COUNT = 4
FILE VERSION COUNT = 3
231
PACKAGE CLUSTER: PKG R WITH FILE X
233
PKG R 3.0
FILE X 2.1
235
PKG R 3.1
FILE X 2.1
237
PKG R 4.1
FILE X 2.1.2
239
PKG R 4.2
FILE X 2.3

FIG. 3

301

▼ 050267c47d484852f42e
v3/repos/npm/npmjs.orgprod/v1/jquery/1.8.2/package/lib/node-jquery.js
▼ 0d7896e2bb23f88e26e5
v4/repos/npmnpmjs.org/prod/v1/jquery/1.7.3/package/tmp/jquery.js
▼ 10ce40582bc36f783e5e
v4/repos/npm/npmjs.org/prod/v1/jquery/1.8.3/package/lig/node-jquery.js
▼ 2fb1ecb058395d047281
v3/repos/npm/npmjs.org/prod/v1/jquery/2.1.1-rc1/package/dist/jquery.js
▼ 4023f6ab9d75626b4890
v4/repos/npm/npmjs.org/prod/v1/jquery/1.6.2/package/dist/node-jquery.js
▼ 44c20da858744b96c62a
v3/repos/npm/npmjs.orgprod/v1/jquery/1.6.2/package/dist/node-jquery.js
▼ 48ca0833a10c45eac99e
v3/repos/npm/npmjs.org/prod/v1/jquery/1.11.1/package/src/jquery.js
v4/repos/npm/npmjs.org/prod/v1/jquery/1.11.0-rc1/package/src/jquery.js
▼ 5c09832dfa0091ca87d4
v4/repos/npm/npmjs.org/prod/v1/jquery/3.0.0-beta1/package/dist/jquery.js
▼ 5c7a64d38ef054823af6
v4/repos/npm/npmjs.org/prod/v1/jquery/1.11.0-beta3/package/dist/jquery.js
▼ adc54e7bc8f083114c5e
v4/repos/npm/npmjs.org/prod/v1/jquery/2.12.pagkage/dist/jquery.js
▼ b08cf5128a86aea81cc6
v3/repos/npm/npmjs.org/prod/v1/jquery/2.1.0-beta3/package/src/sizzle/test/jquery.js v3/repos/npm/npmjs.org/prod/v1/jquery/2.1.1-rc2/package/src/sizzle/test/jquery.js v3/repos/npm/npmjs.org/prod/v1/jquery/1.11.1-rc1/package/src/sizzle/test/jquery.js v3/repos/npm/npmjs.org/prod/v1/jquery/1.9.1/package/src/sizzle/test/jquery.js
▼ eca2caa2d31e5311d9d5
v4/repos/npm/npmjs.org/prod/v1/jquery/2.1.0-rc1/package/src/jquery.js v4/repos/npm/npmjs.org/prod/v1/jquery/2.1.0/package/src/jquery.js v4/repos/npm/npmjs.org/prod/v1/jquery/2.1.1-beta1/package/src/jquery.js v4/repos/npm/npmjs.org/prod/v1/jquery/2.1.1/package/src/jquery.js ...

FIG. 4

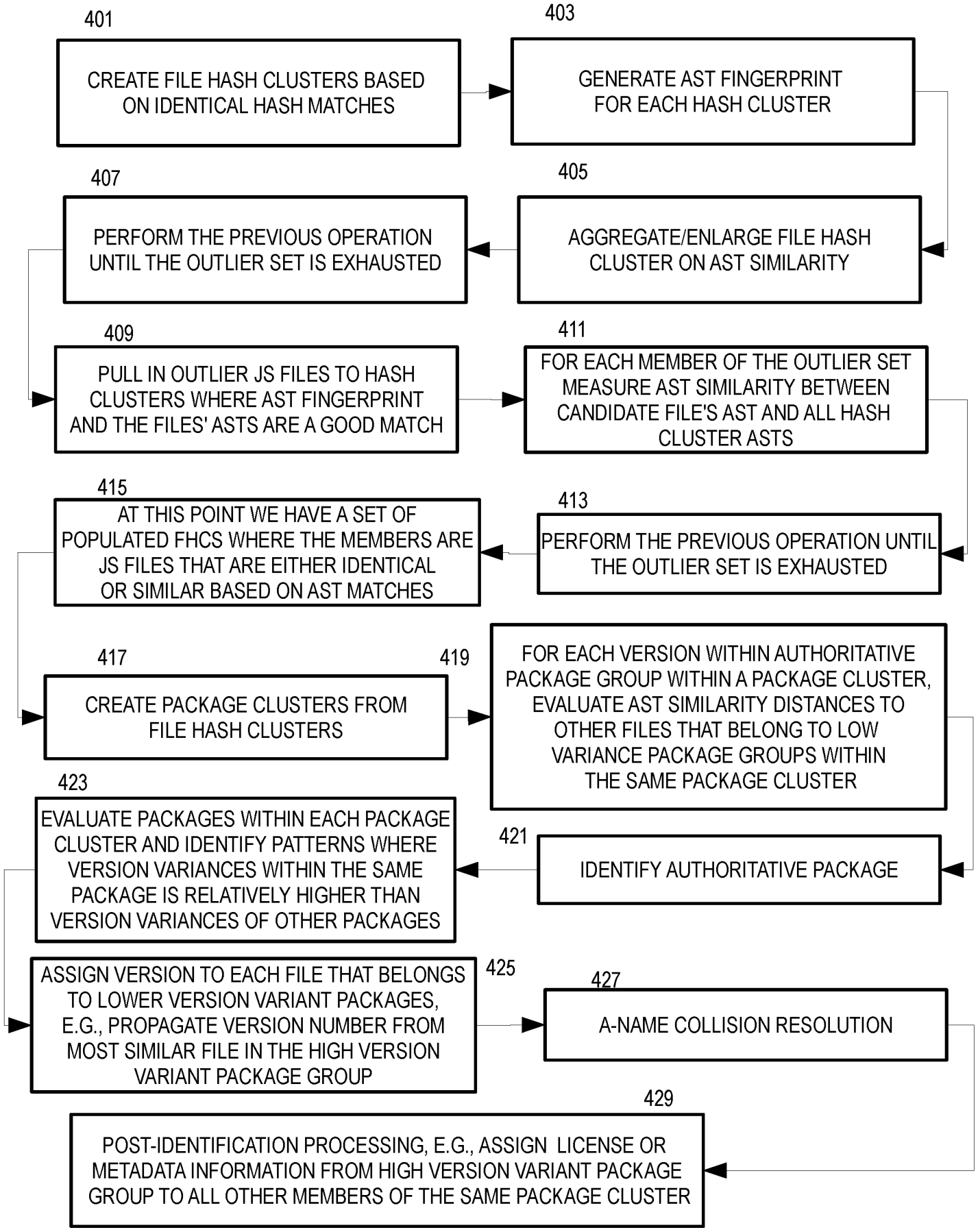

401
CREATE FILE HASH CLUSTERS BASED ON IDENTICAL HASH MATCHES
403
GENERATE AST FINGERPRINT FOR EACH HASH CLUSTER
405
AGGREGATE/ENLARGE FILE HASH CLUSTER ON AST SIMILARITY
407
PERFORM THE PREVIOUS OPERATION UNTIL THE OUTLIER SET IS EXHAUSTED
409
PULL IN OUTLIER JS FILES TO HASH CLUSTERS WHERE AST FINGERPRINT AND THE FILES' ASTS ARE A GOOD MATCH
411
FOR EACH MEMBER OF THE OUTLIER SET MEASURE AST SIMILARITY BETWEEN CANDIDATE FILE'S AST AND ALL HASH CLUSTER ASTS
413
PERFORM THE PREVIOUS OPERATION UNTIL THE OUTLIER SET IS EXHAUSTED
415
AT THIS POINT WE HAVE A SET OF POPULATED FHCS WHERE THE MEMBERS ARE JS FILES THAT ARE EITHER IDENTICAL OR SIMILAR BASED ON AST MATCHES
417
CREATE PACKAGE CLUSTERS FROM FILE HASH CLUSTERS
419
FOR EACH VERSION WITHIN AUTHORITATIVE PACKAGE GROUP WITHIN A PACKAGE CLUSTER, EVALUATE AST SIMILARITY DISTANCES TO OTHER FILES THAT BELONG TO LOW VARIANCE PACKAGE GROUPS WITHIN THE SAME PACKAGE CLUSTER
421
IDENTIFY AUTHORITATIVE PACKAGE
423
EVALUATE PACKAGES WITHIN EACH PACKAGE CLUSTER AND IDENTIFY PATTERNS WHERE VERSION VARIANCES WITHIN THE SAME PACKAGE IS RELATIVELY HIGHER THAN VERSION VARIANCES OF OTHER PACKAGES
425
ASSIGN VERSION TO EACH FILE THAT BELONGS TO LOWER VERSION VARIANT PACKAGES, E.G., PROPAGATE VERSION NUMBER FROM MOST SIMILAR FILE IN THE HIGH VERSION VARIANT PACKAGE GROUP
427
A-NAME COLLISION RESOLUTION
429
POST-IDENTIFICATION PROCESSING, E.G., ASSIGN LICENSE OR METADATA INFORMATION FROM HIGH VERSION VARIANT PACKAGE GROUP TO ALL OTHER MEMBERS OF THE SAME PACKAGE CLUSTER

FIG. 5

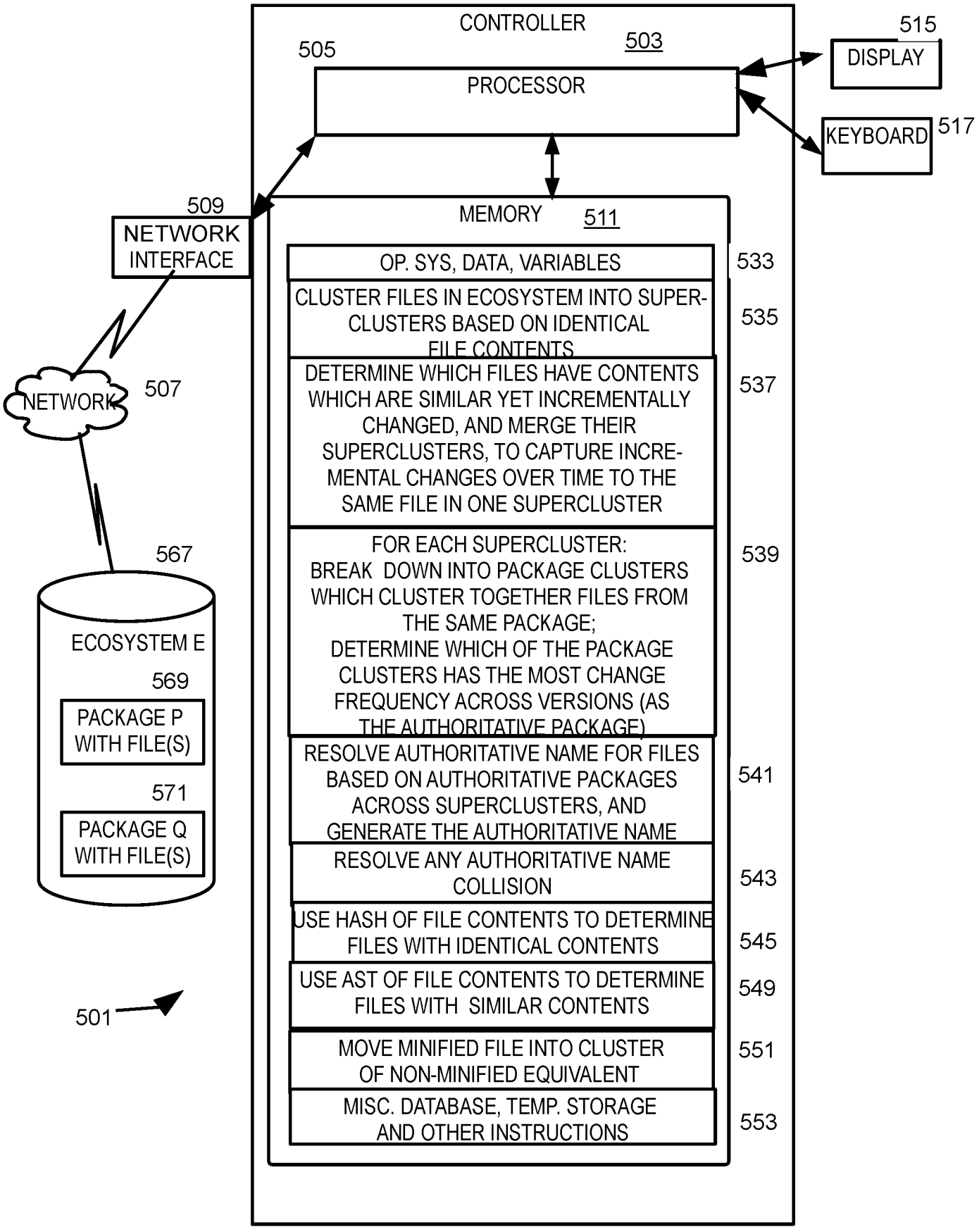

CONTROLLER
503
505
PROCESSOR
515
DISPLAY
KEYBOARD
517
509
NETWORK INTERFACE
MEMORY
511
OP. SYS, DATA, VARIABLES
533
CLUSTER FILES IN ECOSYSTEM INTO SUPER-CLUSTERS BASED ON IDENTICAL FILE CONTENTS
535
DETERMINE WHICH FILES HAVE CONTENTS WHICH ARE SIMILAR YET INCREMENTALLY CHANGED, AND MERGE THEIR SUPERCLUSTERS, TO CAPTURE INCRE-MENTAL CHANGES OVER TIME TO THE SAME FILE IN ONE SUPERCLUSTER
537
FOR EACH SUPERCLUSTER:
BREAK DOWN INTO PACKAGE CLUSTERS WHICH CLUSTER TOGETHER FILES FROM THE SAME PACKAGE;
DETERMINE WHICH OF THE PACKAGE CLUSTERS HAS THE MOST CHANGE FREQUENCY ACROSS VERSIONS (AS THE AUTHORITATIVE PACKAGE)
539
RESOLVE AUTHORITATIVE NAME FOR FILES BASED ON AUTHORITATIVE PACKAGES ACROSS SUPERCLUSTERS, AND GENERATE THE AUTHORITATIVE NAME
541
RESOLVE ANY AUTHORITATIVE NAME COLLISION
543
USE HASH OF FILE CONTENTS TO DETERMINE FILES WITH IDENTICAL CONTENTS
545
USE AST OF FILE CONTENTS TO DETERMINE FILES WITH SIMILAR CONTENTS
549
MOVE MINIFIED FILE INTO CLUSTER OF NON-MINIFIED EQUIVALENT
551
MISC. DATABASE, TEMP. STORAGE AND OTHER INSTRUCTIONS
553
NETWORK
507
567
ECOSYSTEM E
569
PACKAGE P WITH FILE(S)
571
PACKAGE Q WITH FILE(S)
501

FIG. 6

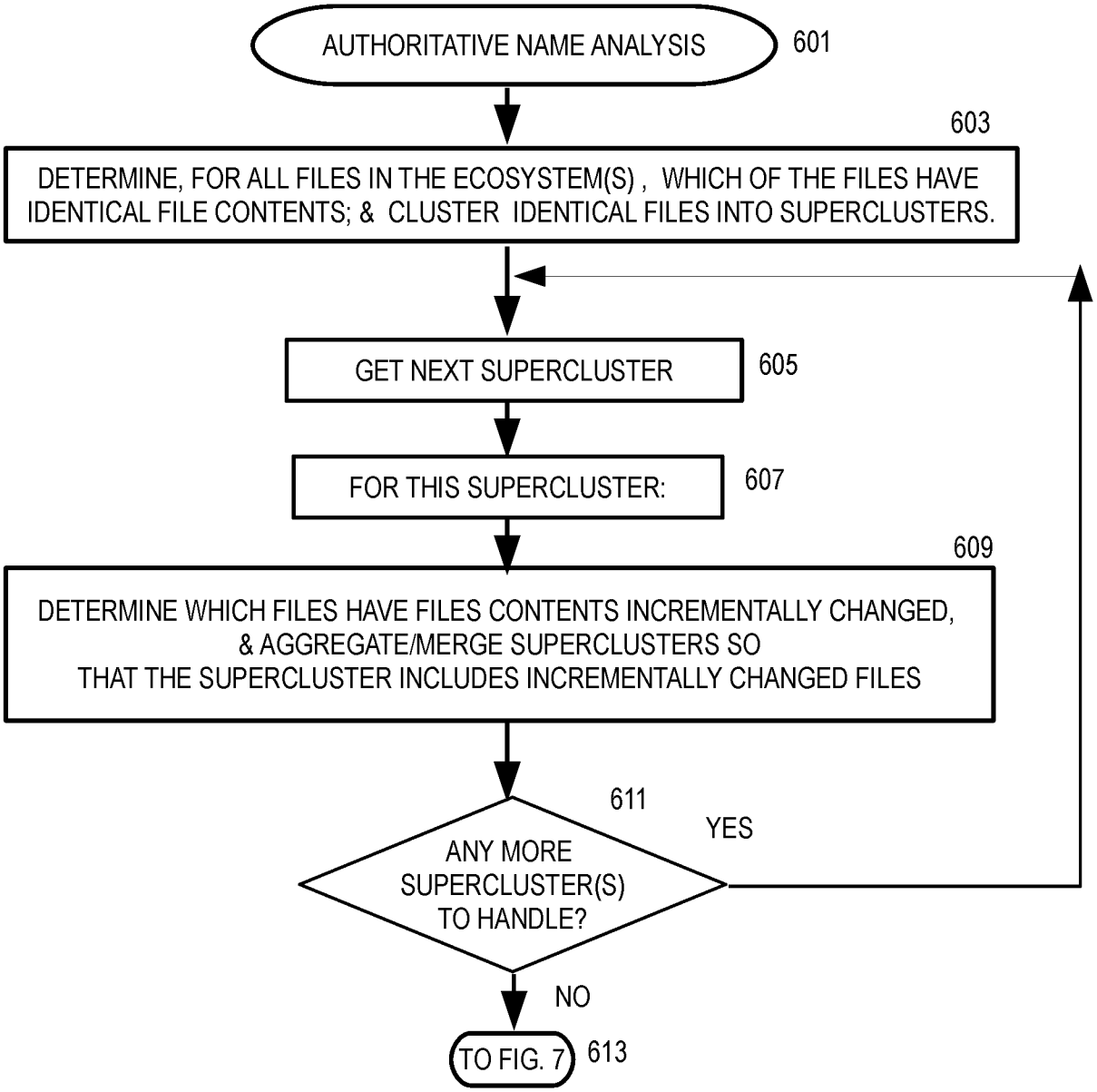

AUTHORITATIVE NAME ANALYSIS
601
603
DETERMINE, FOR ALL FILES IN THE ECOSYSTEM(S) , WHICH OF THE FILES HAVE IDENTICAL FILE CONTENTS; & CLUSTER IDENTICAL FILES INTO SUPERCLUSTERS.
GET NEXT SUPERCLUSTER
605
FOR THIS SUPERCLUSTER:
607
609
DETERMINE WHICH FILES HAVE FILES CONTENTS INCREMENTALLY CHANGED, & AGGREGATE/MERGE SUPERCLUSTERS SO THAT THE SUPERCLUSTER INCLUDES INCREMENTALLY CHANGED FILES
611
ANY MORE SUPERCLUSTER(S) TO HANDLE?
YES
NO
TO FIG. 7
613

FIG. 7

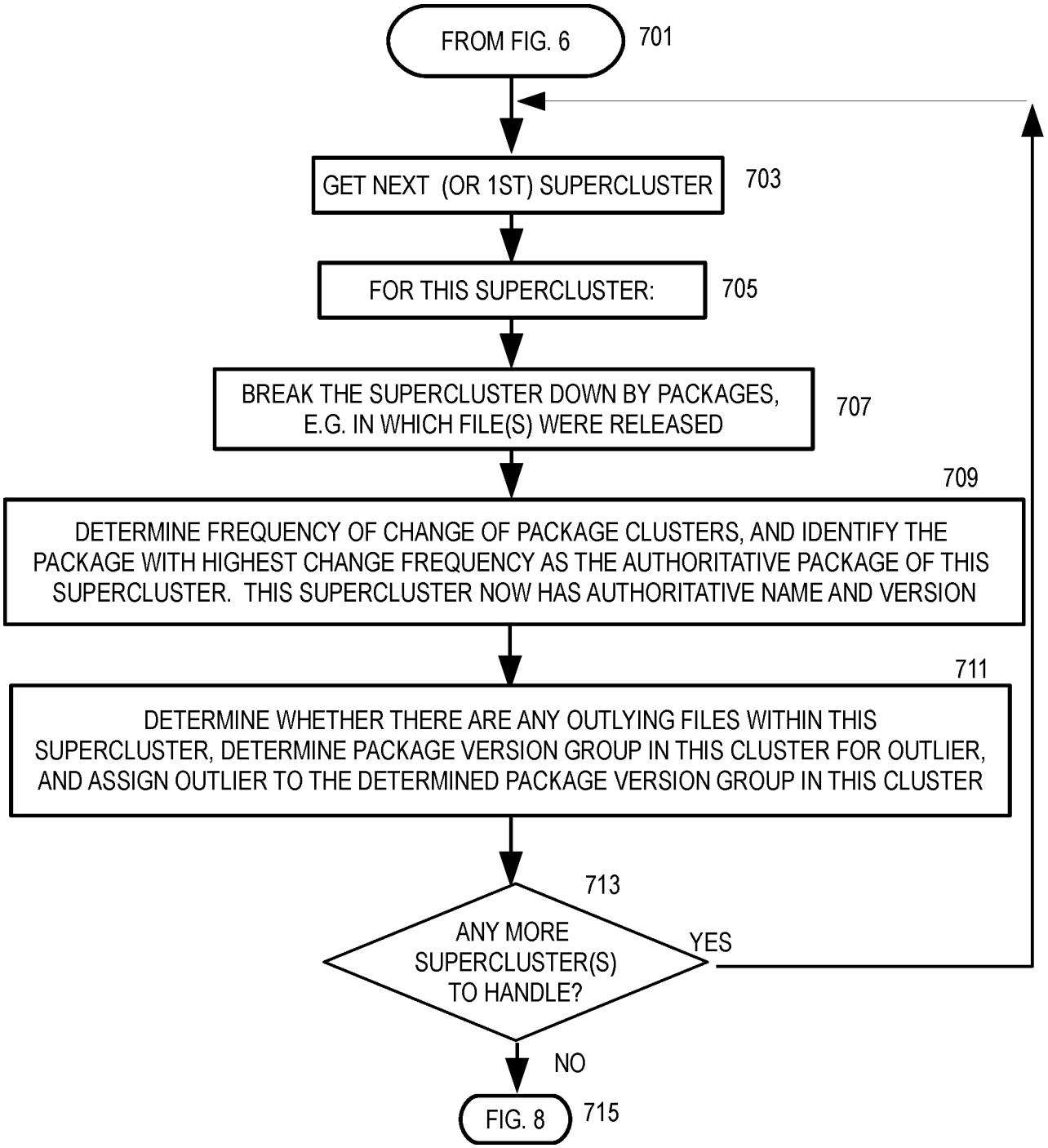
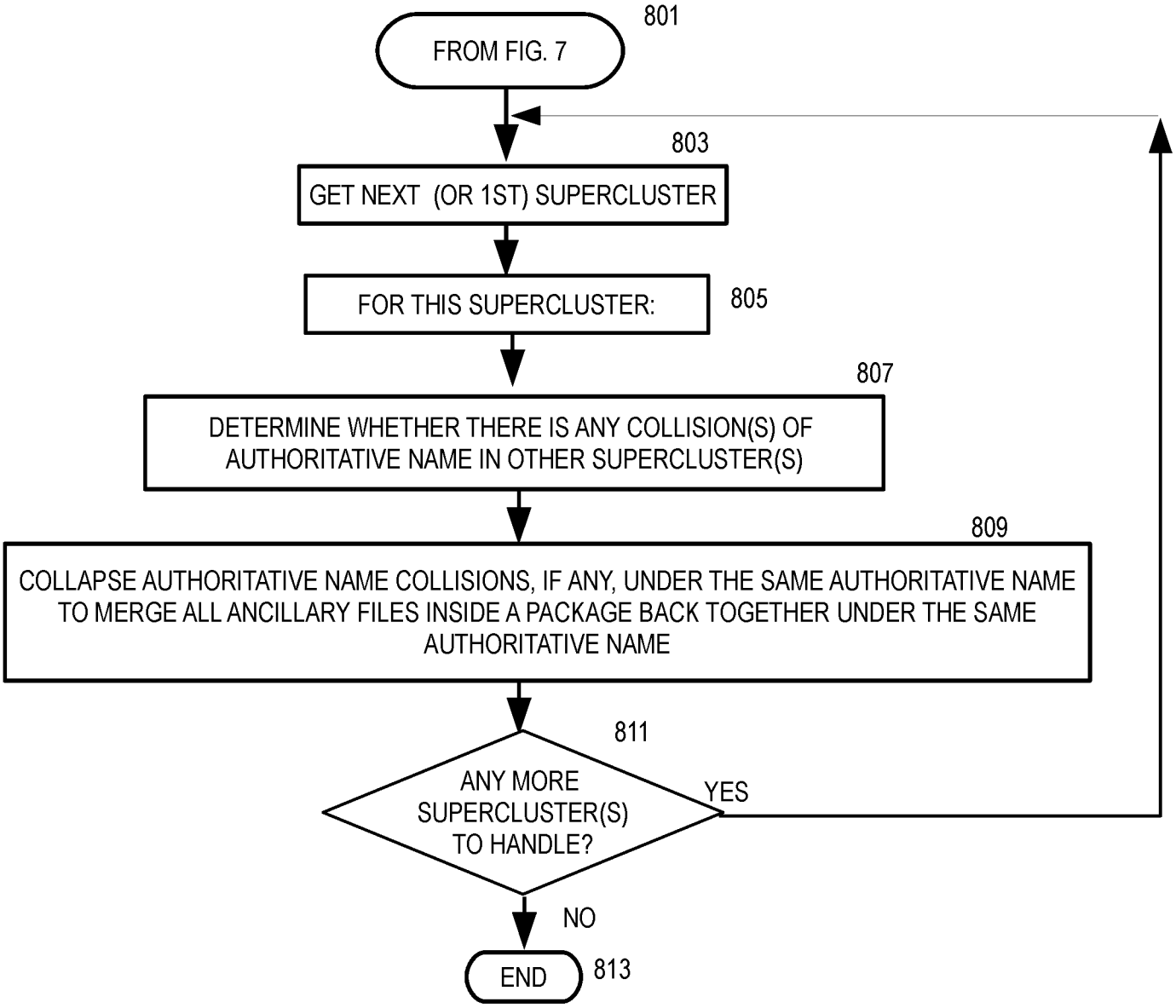
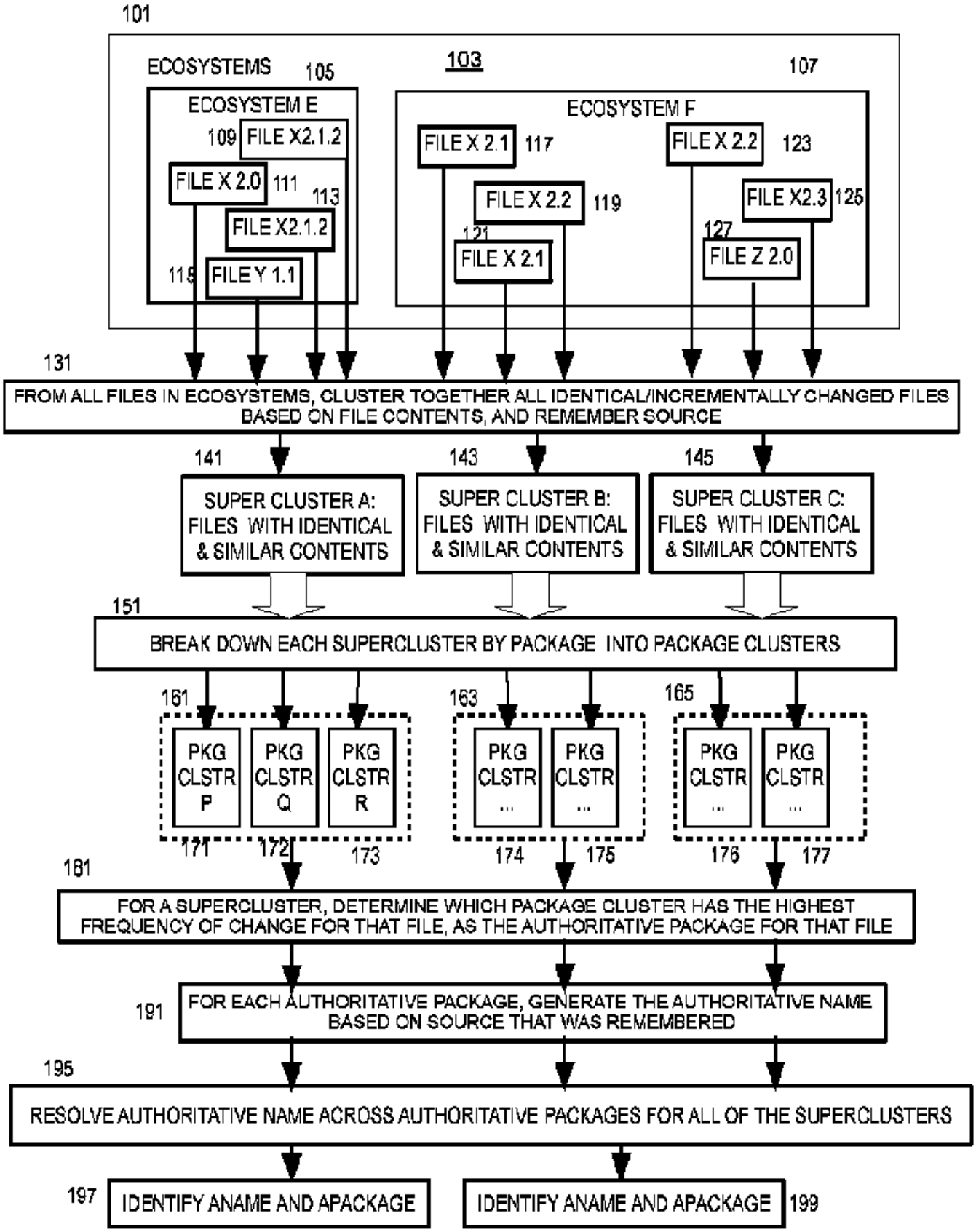

FROM FIG. 6
701
GET NEXT (OR 1ST) SUPERCLUSTER
703
FOR THIS SUPERCLUSTER:
705
BREAK THE SUPERCLUSTER DOWN BY PACKAGES, E.G. IN WHICH FILE(S) WERE RELEASED
707
709
DETERMINE FREQUENCY OF CHANGE OF PACKAGE CLUSTERS, AND IDENTIFY THE PACKAGE WITH HIGHEST CHANGE FREQUENCY AS THE AUTHORITATIVE PACKAGE OF THIS SUPERCLUSTER. THIS SUPERCLUSTER NOW HAS AUTHORITATIVE NAME AND VERSION
711
DETERMINE WHETHER THERE ARE ANY OUTLYING FILES WITHIN THIS SUPERCLUSTER, DETERMINE PACKAGE VERSION GROUP IN THIS CLUSTER FOR OUTLIER, AND ASSIGN OUTLIER TO THE DETERMINED PACKAGE VERSION GROUP IN THIS CLUSTER
713
ANY MORE SUPERCLUSTER(S) TO HANDLE?
YES
NO
FIG. 8
715

FIG. 8


801
FROM FIG. 7
803
GET NEXT (OR 1ST) SUPERCLUSTER
FOR THIS SUPERCLUSTER:
805
807
DETERMINE WHETHER THERE IS ANY COLLISION(S) OF AUTHORITATIVE NAME IN OTHER SUPERCLUSTER(S)
809
COLLAPSE AUTHORITATIVE NAME COLLISIONS, IF ANY, UNDER THE SAME AUTHORITATIVE NAME TO MERGE ALL ANCILLARY FILES INSIDE A PACKAGE BACK TOGETHER UNDER THE SAME AUTHORITATIVE NAME
811
ANY MORE SUPERCLUSTER(S) TO HANDLE?
YES
NO
END
813


101
ECOSYSTEMS
105
103
107
ECOSYSTEM E
ECOSYSTEM F
109
FILE X2.1.2
FILE X 2.0
111
113
FILE X2.1.2
115
FILE Y 1.1
FILE X 2.1
117
FILE X 2.2
123
FILE X 2.2
119
FILE X2.3
125
121
FILE X 2.1
127
FILE Z 2.0
131
FROM ALL FILES IN ECOSYSTEMS, CLUSTER TOGETHER ALL IDENTICAL/INCREMENTALLY CHANGED FILES BASED ON FILE CONTENTS, AND REMEMBER SOURCE
141
143
145
SUPER CLUSTER A: FILES WITH IDENTICAL & SIMILAR CONTENTS
SUPER CLUSTER B: FILES WITH IDENTICAL & SIMILAR CONTENTS
SUPER CLUSTER C: FILES WITH IDENTICAL & SIMILAR CONTENTS
151
BREAK DOWN EACH SUPERCLUSTER BY PACKAGE INTO PACKAGE CLUSTERS
161
163
165
PKG CLSTR P
PKG CLSTR Q
PKG CLSTR R
PKG CLSTR ...
PKG CLSTR ...
PKG CLSTR ...
PKG CLSTR ...
171
172
173
174
175
176
177
181
FOR A SUPERCLUSTER, DETERMINE WHICH PACKAGE CLUSTER HAS THE HIGHEST FREQUENCY OF CHANGE FOR THAT FILE, AS THE AUTHORITATIVE PACKAGE FOR THAT FILE
191
FOR EACH AUTHORITATIVE PACKAGE, GENERATE THE AUTHORITATIVE NAME BASED ON SOURCE THAT WAS REMEMBERED
195
RESOLVE AUTHORITATIVE NAME ACROSS AUTHORITATIVE PACKAGES FOR ALL OF THE SUPERCLUSTERS
197
IDENTIFY ANAME AND APACKAGE
IDENTIFY ANAME AND APACKAGE
199