Title	ノード間連携による情報検索システムの提案
Sub Title	A proposal on an information search system : with nodes collaboration
Author	酒井, 慎一(Sakai, Shinichi) 砂原, 秀樹(Sunahara, Hideki)
Publisher	慶應義塾大学大学院メディアデザイン研究科
Publication year	2009
Jtitle	
JaLC DOI	
Abstract	
Notes	修士学位論文. 2009年度メディアデザイン学 第18号
Genre	Thesis or Dissertation
URL	https://koara.lib.keio.ac.jp/xoonips/modules/xoonips/detail.php?koara_id=KO40001001-00002009-0018

KMD-80835310

修士論文

ノード間連携による情報検索システムの提案

酒井　慎一

2010年1月15日

慶應義塾大学大学院
メディアデザイン研究科

本論文は慶應義塾大学大学院メディアデザイン研究科に
修士 (メディアデザイン学) 授与の要件として提出した修士論文である.

酒井　慎一

指導教員：

砂原 秀樹 教授　（主指導教員）

稲蔭 正彦 教授　（副指導教員）

審査委員：

砂原 秀樹 教授　（主査）

稲蔭 正彦 教授　（副査）

大川 恵子 教授　（副査）

ノード間連携による情報検索システムの提案*

酒井　慎一

内容梗概

インターネット上には膨大な情報があふれてきており，それらを効率よく見つけ出すための情報検索システムが重要となる．そのため，スケーラビリティや情報のマネージャビリティを考慮するとともに検索応答時間や情報の網羅性を確保する必要がある．現在主に利用されている集中インデックス型情報検索システムでは、スケーラビリティや情報のマネージャビリティの確保が困難である．一方，スケーラビリティを考慮したP2P型の情報検索システムの開発も試みられているが検索応答時間や情報の網羅性の確保に課題が残っている．

そこで，本研究ではスケーラビリティや情報のマネージャビリティを考慮しつつ，検索応答時間や情報の網羅性を確保し，より有用な検索結果を提供できる次世代型情報検索システムを構築する。提案手法では，非構造化オーバーレイによるピュアP2Pモデルおよびスーパーノードモデルを基礎モデルとし，各ノードごとに全文検索エンジンを持ちそれらの相互連携で情報検索を実現する．P2P型とすることでスケーラビリティおよび情報のマネージャビリティを確保し，3種類のインデックスを形成することで検索応答時間，情報の網羅性を確保する．また，動的経路構成によって多くのユーザが検索するキーワードについて検索効率を向上させる．

提案システムの有用性を確認するため各検索手法の検索応答時間と情報の網羅性の評価，トラフィック量の算出，動的経路構成における検索効率向上の有効性を確認する評価を行った．評価を行った結果，検索応答時間および情報の網羅性を確保し，また，検索効率を向上が実現できていることが明らかとなった．

*慶應義塾大学大学院 メディアデザイン研究科 修士論文, KMD-80835310, 2010年1月15日.

キーワード

情報検索， P2P， スケーラビリティ， 検索応答時間， 情報の網羅性， 検索効率

A Proposal on an Information Search System with Nodes Collaboration *

Shinichi Sakai

Abstract

Information search system is more important than ever. because information data are increasing on the Internet rapidly. So information search system should have quick response time and high coverage with high scalability and high manageability. But it is impossible for centralized search system to maintain scalability and to provide manageability to each web server, on the other hand, P2P search sytem prior to scalability and manageability has problem about response time and coverage.

In this paper, we propose the next generation information search system which has quick response time, high coverage with scalability and high manageability. In our approach way, each node has full text search engine,and we can get search result by collaboration of each node based on pure P2P model and super node model of unstructured overlay network. P2P network model can bring scalability and manageability for this system, and 3 kind of indexes can bring quick response time and high coverage. And dynamic topology mechanism can improve efficiency of search.

We evaluated this system about coverage and response time with each searching way, improving efficiency with dynamic topoligy mechanism. The result of it

*Master's Thesis, Graduate School of Media Design, Keio University, KMD-80835310, January 15, 2010.

reveals that this system can response quickly, with high coverage, high efficiency and keeping scalability and manageability.

Keywords:

information search, P2P, scalability, response time, coverage, efficiency

目　　次

図　目　次

表　目　次

第1章
序　　論

現在，インターネット上には情報があふれている．しかも，情報はますます増えて行く一方である．その膨大な情報に対し検索へのニーズは今までも高かったもののますます高まってきている．しかも，単純にキーワードマッチさせるだけではなく，ユーザの曖昧なキーワード表現に対していかに検索結果を提示するか，よりユーザの嗜好にあった検索結果を提示するかなど，より高度なニーズが求められるようになってきている．

そのような状況の中で，情報検索システムには増え続ける情報に対して持続的にサービスを提供し続けることが第一に求められる．また，情報提供者が削除した情報がすぐに反映されるような情報のマネージャビリティの高い仕組みでなければならない．さらに例え扱う情報がどれだけ増えたとしても検索応答時間が長くなってしまうわけにはいかない．検索応答時間の短縮を追求する中で検索対象の情報を絞ってしまうことなく網羅性を維持しなくてはならない．

現在主流である集中インデックス型の情報検索システムは，今後も継続的にスケーラビリティを確保するのは困難である．さらに情報のマネージャビリティを確保するのも困難である．一方でスケーラビリティや情報のマネージャビリティの確保可能な P2P 型の情報検索システムは，検索応答時間の確保が困難である．P2P 型の中でも構造化オーバーレイ手法は網羅性が確保できるものの，全文検索の実現が困難である．一方非構造化オーバーレイ手法は全文検索など柔軟な検索が可能である一方で情報の網羅性の確保が困難である．さらに，P2P 型のピュア P2P モデルでは，検索応答時間の確保が困難である．P2P 型のスーパーノードモデルでは，スーパーノードにインデックスを集中させているためピュア P2P モデルに比べ検索応答時間・情報の網羅性が改善できるものの，スーパーノード群の

ノード数が増えれば増えるほどピュア P2P モデル同様，検索応答時間の確保が困難になる．つまり P2P 型ではスケーラビリティは確保できるものの，モデルによってメリットデメリットが混在している状況である．

そこで，本研究は次世代型情報検索システム基盤として P2P 型モデルを改良した情報検索システムを提案する．そのシステムがスケーラビリティや情報のマネージャビリティを考慮するとともに短い検索応答時間で高い情報の網羅性を確保し，常に検索効率を向上していくことのできる仕組みを持つことが目的である．次世代情報検索システムとしては，オーバーレイネットワークの構成方法を検討するシステム基盤，全文検索エンジン，クエリの表現方法とその解釈方法，結果の視覚化とその活用方法などの構成要素が存在するが本研究ではそのうちのシステム基盤の構築を目的とする．

提案する手法では，ピュア P2P モデルおよびスーパーノードモデルを採用し，各ノードごとに全文検索エンジンを持ち，それらが相互に連携することで情報検索が可能になる．各ノードごとにローカルインデックスを持つことで，スケーラビリティ，情報のマネージャビリティを確保するとともに，隣接ノードインデックスを保持することで検索応答時間を短縮する．さらにスーパーノードには広域インデックスを保持させ，それらスーパーノード群に各ノードが情報検索を行うことで情報の網羅性も確保する．また，動的に経路構成を行うことで，検索効率を向上させることができるものとする．なお，本提案は各ノードがすでに検出されていることを前提としている．

第2章
情報検索システムの現状と課題

本章では，情報検索システムに求められるものを明らかにした上で，現在の様々な情報検索システムを整理し分類する．

2.1. 情報検索システムの理想像

Netcraft 社 [1] の統計 (図 2.1) によれば 2009 年 4 月時点で，世界には約 2 億 3 千の web サイトが存在する．注目すべきは，その増加率である．2006 年 1 月から 2 年間の増加率が 51%に対し，2007 年 1 月からの 2 年間の増加率は 109%であった．指数関数的な増加といえる．また今後もこの傾向は続くであろう．そんな状況の下，増え続ける情報の中からユーザが欲しい情報を見つける手段としての情報検索システムに求められる条件は高まるばかりである．

第一に，増え続ける情報に対して，持続的に情報検索サービスを提供し続けられるスケーラビリティの確保が必要である．情報検索サービスを提供し続けるためには，膨大な情報を整理しメタデータとして保持するインデックス，ユーザが検索のためにアクセスする web サーバなどの処理能力が課題になる．また，処理能力のみならず複数のサーバを連携させるネットワークの構成方法なども課題である．さらに，これらのサービスを提供するために必要なコストも課題の一つである．

第二に，情報提供者（各ノードの管理者）にとって情報のマネージャビリティが高い仕組みが必要である．ここでいう情報のマネージャビリティとは，情報の公開・非公開がすぐにインターネット上に反映できることである．情報提供者にとって情報のマネージャビリティが低い場合，もし公開していた情報が間違って

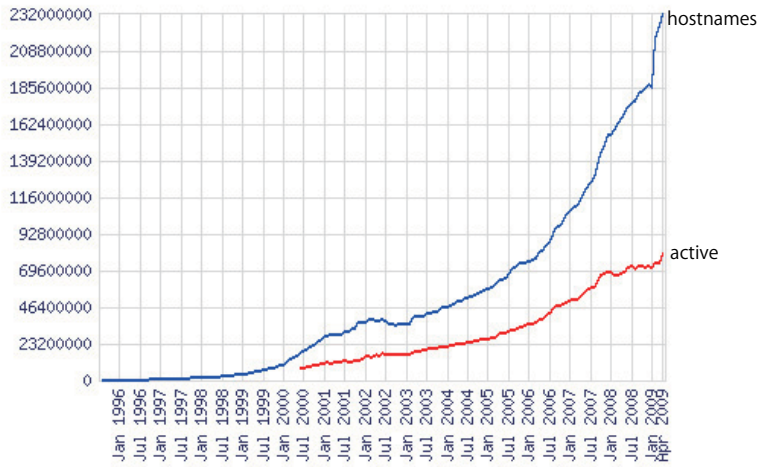

図 2.1 Total Sites Across All Domains August 1995 - April 2009

いたとして削除した時にも検索インデックスのキャッシュにいつまでも残ってしまい，いつまでもアクセスできる状態が続いてしまう．

第三に，情報の検索応答時間は早くあるべきである．例え情報が増加しても検索結果を得るのに 10 分もかかってしまうようなシステムでは実用に耐えない．ユーザは，同じ検索結果を得られるのであればより早い検索応答時間で検索のできるシステムを選ぶ．

最後に，高い情報の網羅性が必要である．情報の増加に伴い，世界中に存在する web サーバから情報をもれなく収集できるかは非常に大きな課題である．

これら 4 つの項目は今までも情報検索システムに求められていたものであるが，今後ますます増加する情報に対してより高次元での実現が必要になるものと考える．

次節では，これらの項目に対し，現在ある情報検索システムがどのような特徴を持つのかを整理し分類する．

2.2. 既存の情報検索システム

2.2.1 集中インデックス型と P2P 型

集中インデックス型

集中インデックス型情報検索システムとは世界中の情報を 1 カ所に集め，1 つのインデックスを形成するシステムである．ユーザは必ず特定サイトに検索要求を行い，すべての応答はその特定サイトから提供される．現在，Google[2] など多くの情報検索サービスはこの方式を採用しており広く普及しているモデルである．以下，特徴をまとめる．

- スケーラビリティ
  集中インデックス型情報検索システムは世界中の情報を 1 カ所に集め，それをインデックス化する．サービスを提供しているサイトのバックエンドに存在するデータベースは巨大なものになる．そのため，分散ファイル技術 [3] や，分散ストレージ技術 [4]，分散ロックシステム技術 [5] などの研究が盛んに行われている．またユーザがアクセスするフロントエンドの web サーバ，実際のマッチング処理を行うエンジン部などの処理も負荷分散や並列処理 [6] をさせる等，工夫がなされている．しかしながら，今後も続く情報の爆発的な増加に対してスケーラビリティを保ち続けることが困難である．またこれらのサービスを提供するためのコストも莫大なものが必要であるという問題がある．

- 情報のマネージャビリティ
  情報提供者とって情報のマネージャビリティは後述の P2P 型に比べ低い．そのため，情報提供者（各ノードの管理者）が公開していた情報を非公開にしても検索インデックスのキャッシュには残っておりアクセスは可能である．

- 検索応答時間
  先ほども述べたサーバを並列動作させるクラスタ構成を採用することにより，非常に高速な検索応答時間を実現している．

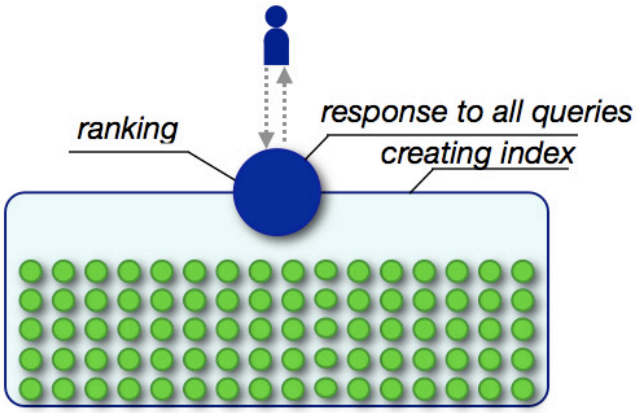

図 2.2 集中インデックス型情報検索システム

- 情報の網羅性
  クローラが世界中の web サーバの情報を集め整理するため網羅性は非常に高い．しかしながら，限られた数のクローラで情報収集を行うため，収集に時間がかかってしまうという問題がある．

P2P 型情報検索システム

集中インデックス型情報検索システムとは違い，特定のノードが世界中の情報を 1 カ所に集め 1 つのインデックスを形成することはなく，それぞれのノードがそれぞれにインデックス情報を保持し，検索時には複数のノードから検索結果を集めてくるシステムである．このシステムは集中型に比べて以下の特徴がある．

- スケーラビリティ
  サーバ機能を各ノードが分散して維持しているためシステムの特定部分に負荷が集中しない．そのため，ユーザの急増や，今後もつづく情報の増加に対しても各ノードのシステムの規模を増加しなくて済む．また，1 つのノードに障害が起きてもそれ以外のノードに及ぼす影響が少ない．さらに，

システムがサーバだけに依存するわけではないため，機器・運用・保守等のコストも情報の増加に比例しない。

- 情報のマネージャビリティ
  それぞれのノードが自ノードのインデックス情報を保持する仕組みのため，集中インデックス型に比べ情報のマネージャビリティは高い．各ノードで情報を削除した場合，すぐにインデックス更新を行えば検索にヒットすることがなくなる．

- 検索応答時間
  情報が分散されているが故に，検索応答時間の短縮が困難である．

- 情報の網羅性
  後述するオーバーレイネットワークの手法によって異なる．

2.2.2 P2P 型におけるオーバーレイの分類

P2P アーキテクチャは物理的構造の上位にあるオーバーレイ・ネットワークにおいて仮想的なネットワークを構成する．この構成の仕方によって構造化オーバーレイと非構造化オーバーレイの 2 つに大別される．

構造化オーバーレイ

オーバーレイ・ネットワークの構成時，構成が数学的なルールに基づくことを仮定したアルゴリズムを活用することにより網羅的な情報探索を実現するものである．そのため，オーバーレイ・ネットワークに存在する全ての情報を対象に探索することができ，100%に近い探索成功率と非常に高い探索効率が特徴である．代表的なモデルに DHT や SkipGraph といったものが挙げられる．DHT モデルでは，P2P 型の弱点である検索応答時間の問題に対して，いくつかの探索経路方法が提案されておりそれらが比較されている [7]．構造化オーバーレイは，キー・ベース・ルーティング (KBR) を用いて探索を行う．キー・ベース・ルーティングとは，各ノードに一意に割り当てられているノード ID とキーとの対応付けで

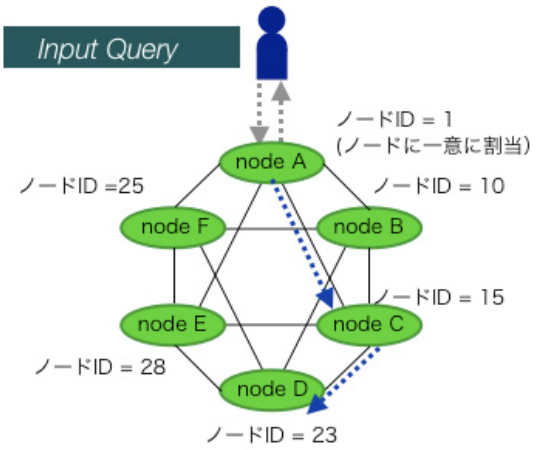

図 2.3 構造化オーバーレイネットワーク

ルーティングを実現するものである．そのため，ノード内の全文を対象とした検索は非常に困難である．クエリのマルチキーワードでの検索を模索している研究 [8] も存在はするが，あくまでクエリ側であって全文検索ではない．DHT モデルで全文検索を試みた研究 [9] もあるが，まず KBR でノードを確定した後に全文検索を行うため，KBR 時点でマッチしなかったノード内の全文検索は行われていない．

非構造化オーバーレイ

非構造化オーバーレイとは検索を行うノードはクエリ・パケットと呼ばれる問い合わせのパケットを，自分と論理的に接続されたノードに対して通知し，隣接のノードは，受け取ったクエリをさらに隣接のノードへとバケツリレーのように中継することで，より広い範囲に検索の要求を広げていくものである．これをクエリフラッディングと呼ぶ．この機構は情報を保持しているノードで検索の処理を行うため，非常に柔軟な検索が可能となる．そのため，構造化オーバーレイでは難しい全文検索も可能である．しかしながら，クエリフラッディングを行う範囲が限られていたり，伝達経路が変化したりする場合はあまり考慮されていない

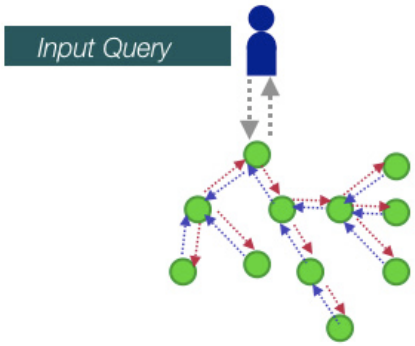

図 2.4 非構造化オーバーレイネットワーク

ため，クエリが全てのノードに行き渡ることは保証されない．そのため，情報の網羅性の確保が困難である．このモデルでは，クエリフラッディングによるトラフィック量の増加が課題になる．そのためトラフィック量をいかに軽減するかという研究がなされている [10][11][12].

2.2.3 P2P 型における 3 つのモデル

インデックス情報の保持の仕方によって P2P アーキテクチャは次の 3 つに大別される．なお，この分類方法は，総務省の分類 [13] に基づくものとする．

ハイブリット P2P モデル

各ノードのメタデータを中央サーバ（インデックスサーバ）と呼ばれるサーバで管理し，コンテンツ自身は各ノードが保有する．ユーザが検索したいコンテンツのキーワードを指定して問い合わせると中央サーバは自身の持つインデックスの中から該当するコンテンツを検索して知らせてくれる．インデックスサーバが提供したノード及びコンテンツ情報に対し，各ノードは直接コンテンツを保持しているノードからコンテンツを取得する．これはつまり前述した集中インデックス型情報検索システムのことである．

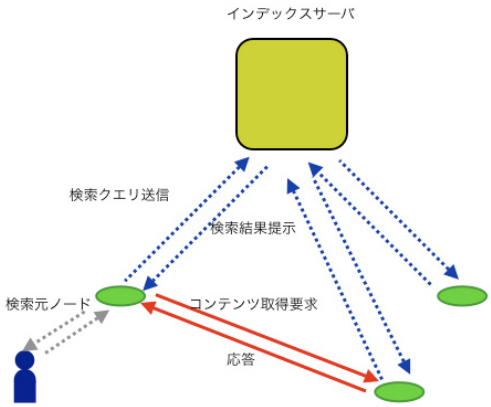

図 2.5 ハイブリッド P2P モデル

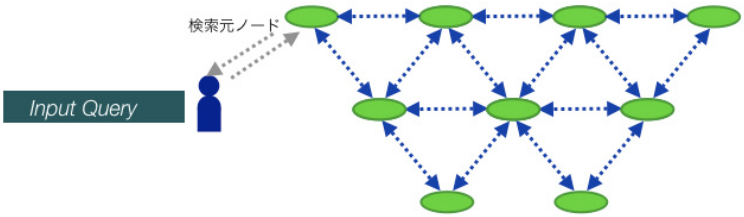

図 2.6 ピュア P2P モデル

ピュア P2P モデル

ハイブリッド P2P モデルとは違い，中央サーバは存在しない．各ノードは同一の機能を保持しノードとノードだけで直接通信を行う．各ノードは自ノードのコンテンツとともに，インデックスも保持し，ノード間で連携して情報検索を実現する．このモデルでは，検索時のクエリのルーティングを効率的に行うことで，情報検索の効率の向上が見込める．そのため，クエリのルーティングをノードの保持している情報によってセマンティックに決定しようという研究も盛んである [14][15][16][17]．また，仮想的なネットワーク構成を可変的に扱うことにより，検索結果の有用性を高めようとする研究も多い [18]．

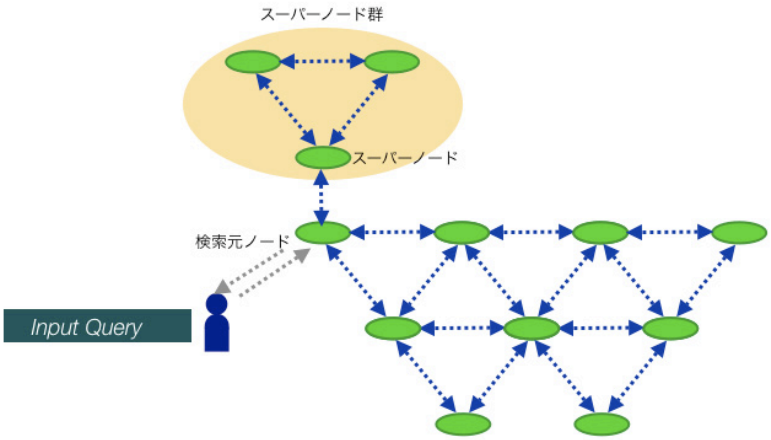

図 2.7 スーパーノードモデル

スーパーノードモデル

ピュア P2P モデルを変形したモデルである．全てのノードが同一機能を持つピュア P2P とは異なり，スーパーノードと言われる高度な処理を行い特別なデータを管理するノードがネットワークに参加していることが特徴である．スーパーノードは処理能力や接続されているネットワークの帯域によって自律的に選出をされ，スーパーノード群を形成する．それらが連携することによってシステムとしてピュア P2P モデルよりも高度なサービスを提供しようとするものである．情報検索システムでは，このスーパーノードがハイブリッド P2P モデルのインデックス・サーバと同様な機能を持つ場合がある．このモデルでは，ピュア P2P モデルでは困難である情報の網羅性の改善と，検索応答時間の改善を目的とした研究 [19] などがなされている．なお，スーパーノードモデルでは一台のスーパーノードが大きくなればなるほど，集中インデックス型に近づくため，ピュア P2P モデルのメリットである情報のマネージャビリティの確保がやや困難となる．

表 2.1 各情報検索システムの特徴 1

特徴	集中型	P2P 構造化オーバーレイ	P2P 非構造化オーバーレイ
スケーラビリティ	×	○	○
検索応答時間	○	×	×
情報の網羅性	○	○	×
マネージャビリティ	×	△	○
全文検索	○	×	○

表 2.2 各情報検索システムの特徴 2

特徴	集中型モデル	ピュア P2P モデル	スーパーノードモデル
スケーラビリティ	×	○	○
検索応答時間	○	×	△
情報の網羅性	○	×	△
マネージャビリティ	×	○	△
全文検索	○	○	○

2.3. まとめ

本章では，情報検索システムの理想像を明らかにした上で，現在の様々な情報検索システムを整理し分類した．各モデルを表にまとめたものが表 2.1 及び表 2.2 である．なお，表中の×○△は 3 つの手法，モデルを相対的に比較した優劣を表している．

集中インデックス型の長所は検索応答時間，情報の網羅性であるが，スケーラビリティ確保，情報提供者にとっての情報のマネージャビリティ確保は困難である．P2P 型において構造化オーバーレイ手法では，情報の網羅性は高いものの，KBR での検索であるため全文検索は困難である．一方，非構造化オーバーレイ

手法では，スケーラビリティは高いものの，検索応答時間，情報の網羅性で問題がある．また，P2P 型においてピュア P2P モデルはスケーラビリティが高く，情報のマネージャビリティの確保は容易であるものの，検索応答時間の確保が困難である．情報の網羅性，検索応答時間を改善させたスーパーノードモデルはピュア P2P モデルに比べ情報のマネージャビリティの確保が困難である．

以上のことより非構造化オーバーレイ手法によるピュア P2P モデルと，スーパーノードモデルのメリットを両方活かせる仕組みがあれば，スケーラビリティや情報のマネージャビリティを確保しつつ，検索応答時間，情報の網羅性の高いものが得られると考える．スケーラビリティ確保や情報のマネージャビリティ確保には，集中インデックス型に比べこれらのモデルが有効である．検索応答時間の短縮には自ノードのインデックスだけでなく隣接ノードの情報もインデックスとして保持することが有効であると考える．また，網羅性の向上にはスーパーノードによる広域のインデックスを形成することが有効であると考える．さらに，動的に経路を構成することが，検索効率を向上させるのに有効であると考える．

次章では以上のことを実現するためのシステムを提案する．

2.4. 補足：情報検索システムで用いられる用語定義

本システムを通じて用いる基本的な用語を下記の通り定義する．なお，それぞれの機能における用語は各該当章にて定義するものとする．

- ノード：本システムに参加する web サーバのこと．本システムのアプリケーション，全文検索エンジンが搭載されているものとする．
- クエリ：検索キーワード等，検索時にノードに送信される情報の全てを指す．
- ホップ：ノードからクエリを送信すること．
- 検索元ノード：ユーザが検索キーワードを入力する任意の web サーバのこと．各ノードから検索結果を収集しユーザに対し提示する．
- ホップ先ノード：クエリを受信したノードのこと．

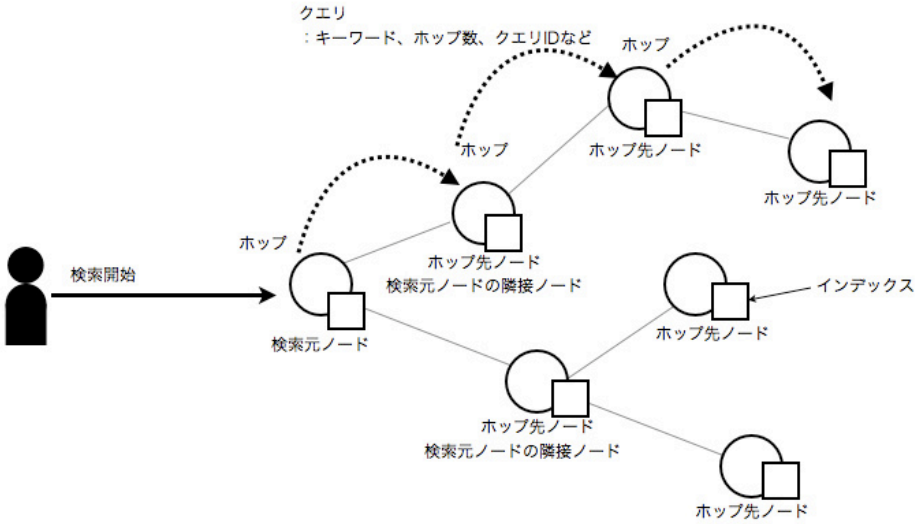

図 2.8 情報検索システムの各用語

- 隣接ノード：あるノードのクエリ送信先ノードリストに登録されており，直接クエリが送信されるノードのこと．
- インデックス：対象となるノードのコンテンツ情報のメタデータファイルのこと．

第3章
ノード間連携による情報検索システムの提案

3.1. 本システムの概要

前章で整理分類した各情報検索システムのモデルの中でP2P型のピュアP2Pモデルおよびスーパーノードモデルを元にしたシステムを提案する．本システムはwebサーバを1ノードとし各ノードが全文検索エンジンを搭載して，相互に連携することでユーザに検索結果を提供できる情報検索システムである．

このシステムによって，各ノードそれぞれがインデックスを保持することになり，1つだけのノードが巨大なインデックスを保持する必要がなくなる．さらにユーザはそれぞれ好きなノードに対し検索を実行するため，1つのノードだけに検索アクセスが集中することもなくなり，スケーラビリティの確保が可能になるものである．また，集中インデックス型でないため情報提供者にとって情報のマネージャビリティを確保しやすい．さらに，検索応答時間の短縮を目的として隣接ノードの情報をインデックスとして保持させる．また，情報の網羅性の確保を目的としてスーパーノードに広域インデックスを保持させ，各ノードはスーパーノードに対してもクエリ送信を行うものとする．なお，スーパーノードは処理能力に余裕のあるノードが自律的にスーパーノードとして機能する．さらに，動的経路構成と優先度に基づくクエリ送信の機能を持たせ，多くのユーザが検索するキーワードに対してより効率的な情報検索を実現し，今後より高度なアルゴリズムを搭載できる基盤とする．

本システムはより多くのノードが参加することで情報検索システムとして有効

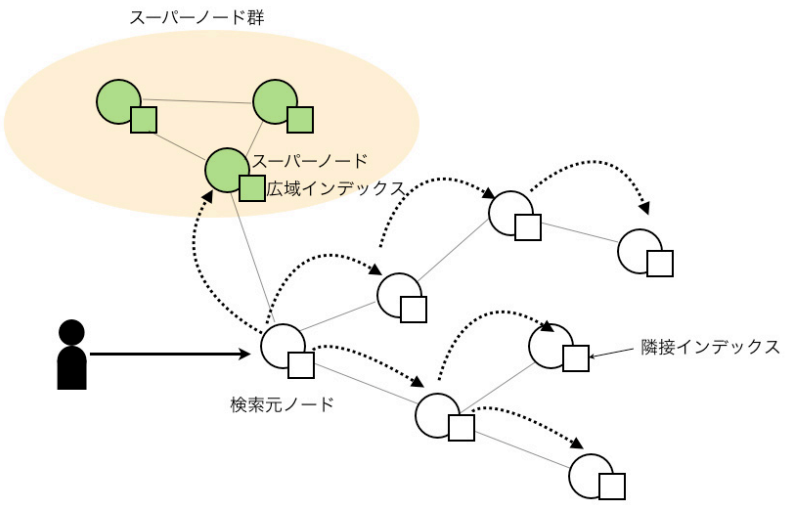

図 3.1 本システムの概要

性が高いものとなる．そのため本研究ではまず各ノードにインストールするためのアプリケーションとして開発することとし，今後の普及，改良に繋げるものとする．

なお、今回各ノードは web サーバのため，PC がノードの場合よりも遥かにノードの離脱は少ない．本研究ではまずノードの離脱はないものと仮定して議論を進める．

3.2. ノード間でのクエリ送受信

本システムでは，スケーラビリティを確保するため，各 web サーバに全文検索エンジンを搭載させる．そしてそのノード間でクエリを送受信して検索結果を広く集めている．ノード間でのクエリ送受信手順を以下に示す．

1. ユーザは検索元ノードとなる web サーバに検索キーワード及びホップ数を入力する．

2. 検索キーワード及びホップ数を入力された検索元ノードは自動でクエリ ID を付与する．

3. 検索元ノードは自ノードの保持しているインデックスに対し全文検索を実行する．

4. 次に検索元ノードは自身の隣接ノードリストから隣接ノードの情報を取り出し，クエリを送信する．

5. クエリを受け取ったホップ先ノードはクエリ ID を元にクエリがループしていないかをチェックし，検索元ノードと同様に，自身の保持しているインデックスに対し全文検索を実行する．

6. さらに自身の隣接ノードリストから隣接ノードの情報を取り出しクエリを送信する．この時，クエリに含まれるホップ数は自身が受け取ったホップ数から 1 を減じたものとする．

7. ホップ数がゼロになるまでクエリをホップさせて行く．

このようにして，複数のノードから検索結果を得ることが可能になる．また，クエリのホップ数を増やせば増やすほど乗数的に対象ノードを増やすことができ，より多くのノードから検索結果が得られ，情報の網羅性は高まる．

この仕組みを利用することで，巨大なインデックスを形成せず各ノード自身のインデックスのみをそれぞれのノードが形成するだけで，理論上すべての情報への検索が可能になる．

しかしながら，この仕組みだけではホップ数が増えれば増えるほど検索応答時間がかかってしまい，決して実用的なシステムとは言えない．また，クエリの送信先が固定的であるため効率的な検索が実現できていない，なぜならばある検索元ノードでよく検索されるキーワードの検索結果を保持するノードが，検索元ノードから遠い場合，つまりホップ数を多く要する場合，毎回，同経路を通過しなくてはならないからである．そのため，検索応答速度を改善する仕組みや検索結果をその後の情報検索に反映できる仕組みが必要である．以下の節では，それらを実現する仕組みを提案し，それぞれがどのような効果をもたらすのかを説明する．

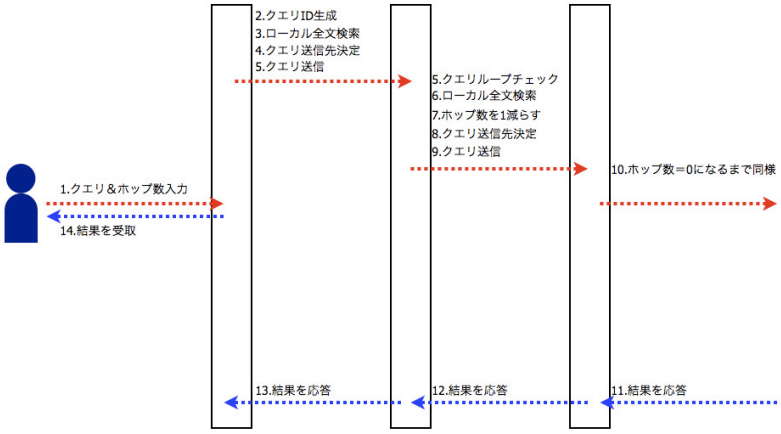

図 3.2 検索の流れ

3.3. インデックス形成

本システムでは，3.1 節で述べた通り，今後も続く爆発的な web データの増加に耐えうるスケーラビリティを確保することを目的として集中インデックス型ではなく，P2P 型を採用している．しかしながら，ホップ数が増えれば増えるほど検索応答時間がかかってしまう．ホップ数を減らせば，情報の網羅性が低下してしまう．そのため，P2P 型のメリットを活かしつつも，検索応答時間の短縮，情報の網羅性の向上を目的として自ノードのコンテンツのみならず複数ノードのコンテンツをインデックス化することを提案する．これは集中インデックス型と P2P 型が対極にあるモデルであるのに対し，それらの中間的なモデルを採用することにより両方のメリットを享受できるものである．具体的には自ノードの情報に対するインデックスに加えて以下の 2 つのインデックス形成を提案する．

- 隣接ノードインデックス：検索応答時間の短縮を主目的に導入する．
- 広域インデックス：情報の網羅性の向上を主目的に導入する．

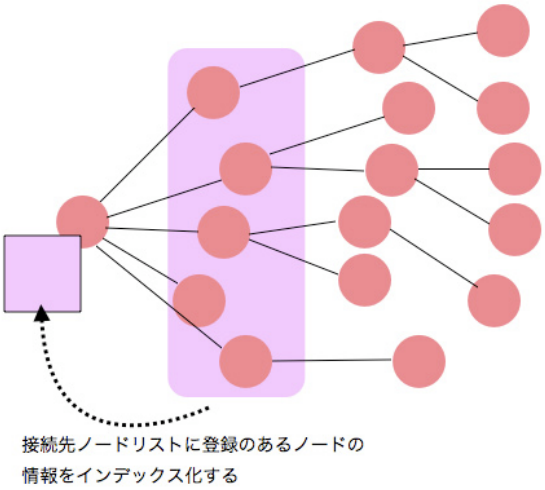

図 3.3 隣接ノードを含むインデックス形成

3.3.1 隣接ノードを含むインデックス形成

各ノードは自身のコンテンツのみならず隣接ノードノードリストに登録のあるノード（隣接ノード）のコンテンツもインデックス対象として収集しインデックス（隣接ノードインデックス）を形成する．検索実行時には，各ノードはそれぞれ隣接ノードインデックスに対し検索を実行し，結果を応答することになる．

この仕組みは動的経路構成，優先度に基づくクエリ送信で得られるメリット（少ないホップ数で多くのヒット数が得られるなど）を保持しつつ，インデックス化することで更なる検索応答時間の改善をもたらす．また，インデックス対象を隣接ノードのコンテンツに限定することで，ノードに対するインデックス形成の負荷も軽減する．

3.3.2 スーパーノードにおける広域インデックス形成

本システムでは，通常のノードに加え，スーパーノードが存在する．処理能力が高く，稼働状況からもまだ処理能力に余裕のあるノードが自律的にスーパーノードとなり，隣接ノードインデックスに加え，より広域なノードのコンテンツを対

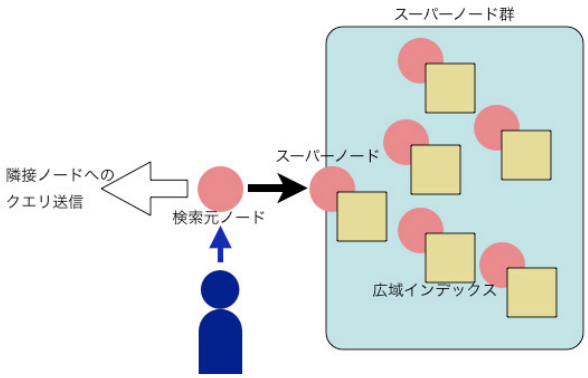

図 3.4 スーパーノードと広域インデックス形成

象として収集しインデックス（広域インデックス）を形成する．各ノードはまず，自ノードから見たクエリの送信経路を表すノードマップを作成する．そのノードマップに存在する任意のスーパーノード 1 台に対してクエリ送信を行う．クエリを受信したスーパーノードは，自身が保持するスーパーノードリストに従い，スーパーノード群へクエリを送信し検索結果を収集する．

スーパーノードへのクエリ送信により検索応答時間，クエリ送信におけるトラフィック量の削減とともに，高い情報の網羅性を確保できる．

また，隣接ノードインデックスとともに，広域インデックスの仕組みを導入することで，冗長性を確保することが可能になる．

例えば，あるノードがシステム障害でダウンした場合，集中インデックス型情報検索システムでは，情報検索そのものができなくなってしまう．また，単純にクエリ送信だけのピュア P2P モデルの場合，他のノードから情報検索を実行すれば良いため，情報検索そのものができなくなってしまうことはないものの，ダウンしたノードの情報は検索が不可能となってしまう．しかしながら，隣接ノードインデックスや広域インデックスを導入することで，たとえあるノードがダウンしたとしても情報検索システムとしての冗長性が確保できることはもちろんのこと，各ノードが重複して周辺ノードの情報をインデックスとして保持していることで，情報の冗長性も確保できることになる．

3.3.3 スーパーノードと一般ノードの存在意義

本システムでは通常のノード（一般ノード）とスーパーノードの両方が必要である．なぜならそれぞれメリットデメリットがあるためである．スーパーノードのみしか存在しない場合，検索応答時間は一般ノードをクエリフラッディングするよりも圧倒的に短く，検索時の応答トラフィックも少ないままで多くの情報に対して検索を実行することができる．つまり検索コストが非常に低く済む．また多くのノードをランダムにインデックス化しているために検索頻度の少ないキーワードに対しても検索結果が得られる可能性が高い．一方，検索元ノードに対し多くのユーザが頻繁に検索するキーワードかどうかは検索結果に反映されない．また，より多くのノードの情報をインデックス化するためどうしても広域インデックスは他の隣接ノードインデックスやローカルインデックスに比べ更新頻度が低くなるため，新しい情報に対しての検索には対応できない．

一般ノードのみしか存在しない場合，スーパーノードのみの場合と比べ格段に検索応答時間はかかってしまう．また検索時の応答トラフィックも多い．つまり，スーパーノードのみの場合と比較して圧倒的に検索コストが高い．さらに，検索元ノードに対し検索されるキーワードの頻度が低かった場合，検索結果を得られにくい．一方，検索元ノードに対し多くのユーザが頻繁に検索するキーワードであれば，後に述べる動的経路構成の効果によって多くの検索結果を得られやすくなる．また広域インデックスに比べてインデックスの更新頻度が高いため，新しい情報に対する検索にも対応可能である．

以上のことよりスーパーノードへの検索は検索コストを抑えつつ様々な検索キーワードに対して検索を行いたい場合に有効であり，一方で一般ノードへの検索は検索コストがかかってしまうものの検索頻度の高いキーワードに対してや新しい情報に対して検索を行いたい場合に有効であるといえる．そのため情報検索システムとしてユーザの様々な検索ニーズを満たすためには両方をある割合で活用する必要がある．

3.4. 動的経路構成

クエリの送信先が固定的な場合，検索結果を反映できずに効率的な情報検索が実現できない．なぜならばある検索元ノードでよく検索されるキーワードの検索結果を保持するノードが検索元ノードから遠い場合でも，毎回，同経路を通過しなくてはならないからであるそのため本システムでは，検索結果に対してユーザがクリックをした場合に各ノードのクエリ送信先を順次追加して行くことで全体として動的に経路を構成し，効率的な情報検索を実現する．追加していく際の加点アルゴリズムについては，今後より高度なアルゴリズムを搭載できるようにし，本システムはその基盤を提供するものとする．動的な経路更新が行われるフローを以下に示す．

ユーザが検索を行い，検索結果を得るところまでは3.2節で説明した通りである．

1. ユーザは得られた検索結果の中で閲覧したいホームページのURIをクリックすることになる．

2. URIがクリックされた際，検索元ノードはクリックされたURIが自身の隣接ノードリストにない場合は登録する．

3. 隣接ノードリストにある場合は，該当URIの点数欄に加点をする．

4. 登録されたノードには次回検索時から他のノードをホップすることなく直接クエリ送信が行われる．

このようにして検索が実行され，ユーザがその結果を活用するたびに，隣接ノードリストの更新が行われる．更新は各ノードでそれぞれ行われており，時間の経過とともに動的に経路が変更されて行く仕組みになっている．

この仕組みは検索応答時間の改善をもたらす．検索が繰り返されることにより，ユーザが実際にクリックした検索結果を保持するノード群が隣接ノードリストに追加されていくため，検索結果を得るまでのホップ数を大幅に削減することが可能である．

また，本機能を導入した結果．あるノードの周辺には近いコンテンツを保持しているノードが集まる可能性が高い．なぜならば，あるノードで検索されるキー

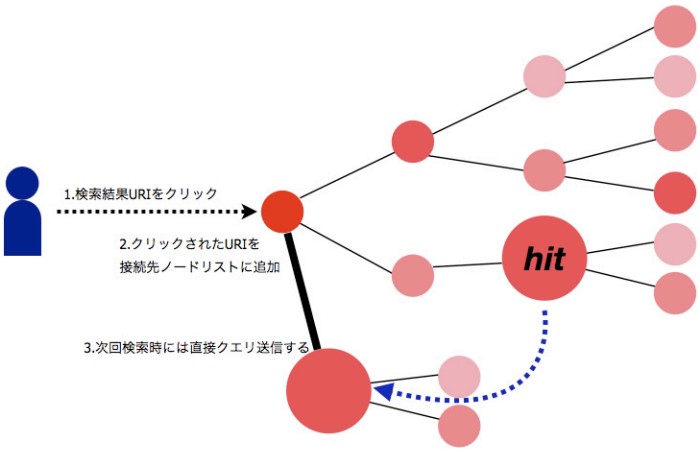

図 3.5 動的経路構成

ワードは多様であるものの，そのノードの情報を閲覧していたユーザが，次に検索したいキーワードはある程度そのノードが保持している情報に内容として近いものが多いと考えられるためである．例えば，草野球チームのホームページが検索元ノードだった場合，やはり野球に関する検索キーワードが多いと考えられる．すると，隣接ノードリストに登録されているノードも野球関連の情報が多いノードになる可能性が高い．さらにそれらノードの隣接ノードリストにも近い情報をもつノードが登録されている可能性が高い．つまり，情報検索が繰り返されることにより情報の近いノード同士が仮想的に近づくことになる．

3.5. 優先度に基づくクエリ送信制御

本システムではクエリ送信先を各ノードの優先度に基づいて決定し，効率的なクエリ送信を実現している．隣接ノードリストにはノードごとに検索後にユーザがクリックした回数に基づく点数情報が付与されている．情報検索時にはその点数情報の高いノードに対してのみ順に一定数クエリを送信する．3.4 節で述べた通り，ユーザが検索結果の URI をクリックすることで，その URI を隣接ノードリストに登録していくが，時間が経つにつれ登録されたノード数は膨大なものと

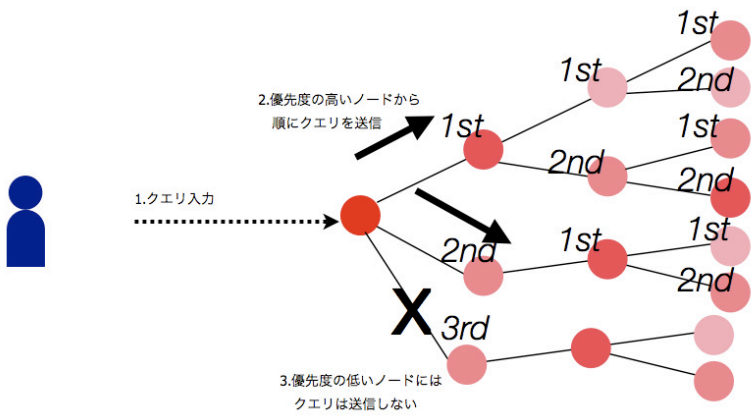

図 3.6 優先度に基づくクエリ送信

なってしまう．そのため，ユーザがURIをクリックした際，ノードの登録とともに点数情報に加点をしていく．またすでに登録のあるノードに関しては加点のみを行う．こうすることで，ユーザがクリックする頻度に応じてノードの点数が決定される．なお，点数情報の低いノードに対してはクエリを送信しない．

この仕組みは検索応答時間，クエリのフラッディングによるトラフィック量を削減する．本システムで採用しているピュアP2Pモデルやスーパーノードモデルはクエリのフラッディングによるトラフィック量の増加が大きな問題となるが，この仕組みで削減することが可能である．

3.6. 提案のまとめ

本章ではノード間連携による情報検索システムの提案を行った．P2P型のピュアP2Pモデルを採用したことでスケーラビリティや情報のマネージャビリティを確保しつつ，ノード間でのクエリ送信を基本とし，検索応答時間短縮のために隣接ノードインデックス，情報の網羅性の向上を目的としてスーパーノードによる広域インデックスの形成を提案した．さらに検索結果を反映させた動的経路構成および優先度に基づくクエリ送信の仕組みを導入することで，随時検索効率を向上する仕組みを提供することができるものとした．

第4章
設　　計

4.1. 設計概要

本システムは以下の処理で構成されている.

- インデックス形成および更新処理
- 検索処理
- 動的経路構成処理

本システムの処理の全体像を図 4.1 に示す．次節からは，設計のための予備実験とそれぞれの処理の詳細を説明する.

4.2. 予備実験

本システムを設計するために，以下の項目について予備実験を行った．なお，評価環境は 6 章における環境と同一とする.

実験内容

実験は以下の 5 つを行った.

- 実験 1：コンテンツ容量変化に伴うインデックス作成時間の測定.
  ローカルインデックス更新周期を決定するためにテキストコンンテンツ容量変化に伴うインデックス作成時間を測定する.

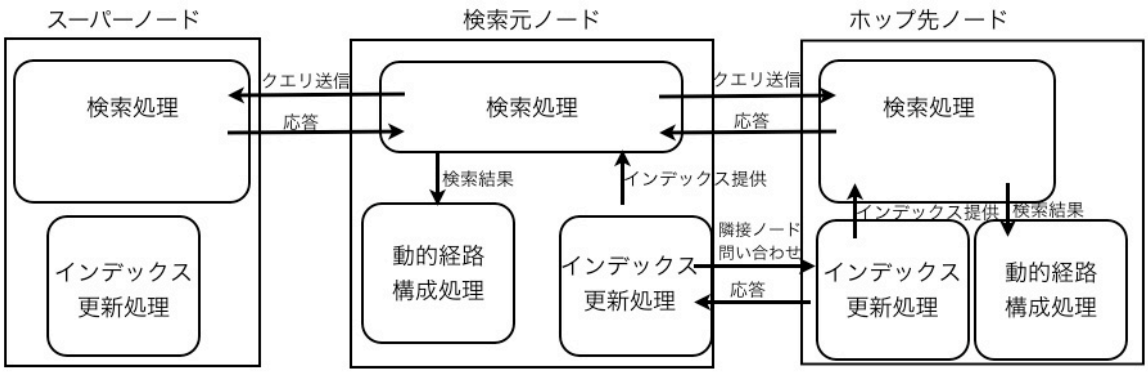

図 4.1 処理の全体像

- 実験 2：コンテンツ容量変化に伴う CPU およびメモリ利用率，インデックスファイルサイズの測定.
  ローカルインデックス作成時のノード負荷，また，ノードの必要条件としてハードディスクの空き容量の必要量を算出するために，テキストコンテンツ容量変化に伴う CPU 及びメモリ利用率，インデックスファイルサイズを測定する.

- 実験 3：対象ノード数変化に伴うインデックス作成時間の測定.
  隣接ノードインデックスおよび広域インデックスの更新周期を決定するために，対象ノード数変化に伴うインデックス作成時間を測定する．なお，対象ノードのコンテンツは全てテキストコンテンツで 100MB とした.

- 実験 4：対象ノード数変化に伴う CPU およびメモリ利用率の測定.
  隣接ノードインデックスおよび広域インデックス作成時のノード負荷，また，ノードの必要条件としてハードディスクの空き容量の必要量を算出するために，対象ノード数変化に伴う CPU およびメモリ利用率，インデックスファイルサイズを測定する．なお，対象ノードのコンテンツは全てテキストコンテンツで 100MB とした.

- 実験 5：クエリ送信台数変化に伴う検索応答時間の測定.
  クエリ送信時の同時送信ノード数を決定するために，クエリ送信台数変化

に伴う検索応答時間を測定する．

実験結果

予備実験の結果を以下にまとめる．

- 実験 1：1 ノード内でのインデックス対象容量 (v_i) とインデックス作成所要時間 (t_i) はほぼ比例関係にあり，以下の式で表すことができる．

$$t_i = 0.175 v_i (秒)$$

- 実験 2：インデックス対象容量が増加しても CPU およびメモリの使用率はほぼ変化がないことが判明した．また 1 ノード内でのインデックス対象容量 (v_2) の変化におけるインデックスサイズ (s_2) の変化は以下の式で表すことができる．

$$s_i = 0.8 v_i + 7 (MB)$$

- 実験 3：インデックス対象ノード数 (n_i) の増加とインデックス作成所要時間 (t_i) はほぼ比例関係にあり，以下の式で表すことができる．

$$t_i = 8.5 n_i (秒)$$

- 実験 4：インデックス対象ノード数が増加したとしても CPU 及びメモリの使用量はほぼ変化がないことが判明した．またインデックス対象ノード数 (n_i) の増加におけるインデックスサイズ (s_i) の変化もほぼ比例関係にあるといえ，以下の式で表すことができる．

$$s_i = 26 n_i + 9 (MB)$$

- 実験 5：直列モデル及び並列モデルとも台数が増えれば増えるほど検索応答時間が長くなっていることが判明した．また，両者にはそれほど大きな差がないことが判明した．

4.3. インデックス形成および更新処理

本システムはローカルインデックス，隣接ノードインデックス，広域インデックスの 3 種類を保持する．本節ではインデックス形成および更新処理についての詳細を説明する．

4.3.1 ローカルインデックス形成および更新

自ノード内のコンテンツを対象にインデックス化したファイルをローカルインデックスという．なお，自動で更新されるものとし，管理者は更新間隔を指定できるものとする．

実験 1 より，1 ノード内でのインデックス対象容量 (v_i) とインデックス作成所要時間 (t_i) は t_i=0.175v_i（秒）で表すことができる．ローカルインデックス作成には対象テキストコンテンツの容量が 1GB であっても約 2 分で完了する．なお，これは対象コンテンツが全てテキストデータの場合である．実際の web サイトは画像や動画も多く，ランダムに 30 サイトのトップページのテキスト比率を調べたところ，ページ全体の容量に対してテキスト容量の割合は 9.61%であった．大規模動画サイトや写真共有サイトではさらにその割合は低下する．なお，インデックス化はテキストデータに対し行われる．そのため実際には 1GB のテキストコンテンツを保持する web サイトは全体としてはさらに巨大な web サイトであり，それでも 2 分でインデックス作成が完了することになる．

また実験 2 より 1 ノード内での対象容量 (v_i) 変化に伴うインデックスサイズ (s_i) の変化は s_i=0.8v_i+7 (MB) で表すことができるため，テキストコンテンツ容量の約 8 割のファイル容量でインデックスファイルが生成される．そのため，多くのサイトの場合ローカルインデックス用に必要なハードディスクの空き容量は約 1GB もあれば十分である．

図 4.2 隣接ノードインデックスの形成及び更新処理

4.3.2 隣接ノードインデックス形成および更新

本システムでは自ノード内のローカルインデックスとは別に隣接ノードのコンテンツをインデックス化し，保持する．これを隣接ノードインデックスとする．検索実行時には，自ノードのインデックスとともに，この隣接ノードインデックスもあわせて利用し，検索を実行する．

なお，動的経路構成により随時対象ノードであるより有用なノードが隣接ノードリストに登録されるメリットを活かすために，短い期間での更新としたい．そのため，3 時間に 1 度の自動更新とし，2 時間以内に更新が完了するもとのする．

1 ノードあたりのテキストコンテンツを 100MB と仮定し，対象ノード数およびインデックスサイズを決定する．実験 3 より対象ノード数 (n_i) 変化に伴うインデックス作成所要時間 (t_i) は t_i=8.5n_i（秒）で表されるため，対象ノード数は 845 ノードとなる．実験 4 よりその際のインデックスサイズ (s_i) は s_i=26n_i+9(MB) で計算でき，約 21.5GB となる．

隣接ノードインデックス形成および更新処理の詳細を図 3.3 に示し，以下，流れを説明する．

1. データベーステーブルの隣接ノードリストから隣接ノードを取得する．
2. 隣接ノードの点数情報が高いものから 845 ノード分を隣接ノードインデックスの設定ファイルに登録する．

3. インデックス更新を実行する．

4. 845 ノード分全てがクロールできた時点で更新完了とする．

4.3.3 ノードマップとスーパーノードの広域インデックス形成および更新処理

検索元ノードは検索実行時に得るクエリ未送信先リストをその都度ノードマップデータベーステーブルに登録する．そのリストを元に 1 日 1 回，ノードマップを作成する．具体的には，リストにあるノードにそれぞれ問い合わせ，各ノードの隣接ノードリストを取得し，ノードマップデータベーステーブルに登録し，さらにその登録したノードに対し，問い合わせをして行く．これによりそのノードを起点としたある時点での経路が一覧で見られるようにできる．

また，それぞれのノードは，自身の隣接ノードリストを応答する際に，同時に自身がスーパーノードであるかどうかもあわせて応答する．応答を受け取った問い合わせ元のノードは，スーパーノードのフラグがあれば，そのノードをデータベーステーブルのスーパーノードリストに加える．

なお各ノードは，ノードマップ作成時に以下の条件を満たす場合，自律的にスーパーノードとして機能することになる．以下の条件をスーパーノード条件とする．

- 過去 1 時間の平均 CPU 使用率が 50%以下であること
- 過去 1 時間の平均メモリ使用率が 40%以下であること
- ハードディスクの空き容量が 1,200GB 以上あること．

スーパーノードになったノードは以下の機能を有する．

- スーパーノードリストに基づくスーパーノード群へのクエリ送信
- 広域インデックスの形成と更新

スーパーノードになったインデックス作成は120時間(5日間）に1度とし，100時間以内に更新が完了するものとする．ノードごとにコンテンツ容量が違うため，明確に100時間でインデックス化できるノード数は特定できない．しかしながら目安として実験3より算出ができる．対象ノード数 (n_i) 変化に伴うインデックス作成所要時間 (s_i) は s_i=8.5n_i（秒）で表されるため，1ノードあたり100MBのテキストコンテンツがあると仮定した場合の対象ノードは42,250ノードとなり，その際のインデックスサイズ (s_i) は s_i=26n_i+9(MB) で計算でき，約1,075GBとなる．また，3.4節で述べた通り，動的経路構成により自ノードの近隣にはユーザが検索する頻度の高いキーワードにヒットするノードが集まっている可能性が高く，特定のクエリに対しては効率的に検索が可能になるものの，一方で多様なクエリに対しては結果を提供しにくい．そのため多様なクエリに対しての結果を提供したい広域インデックスの対象ノードは，自ノードからのホップ数が大きくあるべきである．以上の理由から，ノードマップに登録のあるホップ数の1番大きいノード群を対象ノードとして，そのノードを起点にリンクをたどる形でのインデックス形成とする．処理の詳細を図4.3に示し，以下，流れを説明する．

1. データベースのノードマップテーブルのホップ数1のノードを取得する．
2. 取得したノードに対し，それぞれの隣接ノード5件を問い合わせる．
3. 応答のあった隣接ノードをホップ数2としてノードマップテーブルに登録する．
4. その際，登録したノードがスーパーノードの場合はスーパーノードリストテーブルにも登録する．
5. 登録したホップ数2のノードに対し隣接ノード5件を問い合わせる．
6. ホップ数20まで繰り返す．
7. 自ノードがスーパーノードならば広域インデックスを更新する．
8. スーパーノードでない場合，スーパーノード条件を満たすかどうかをチェックする．

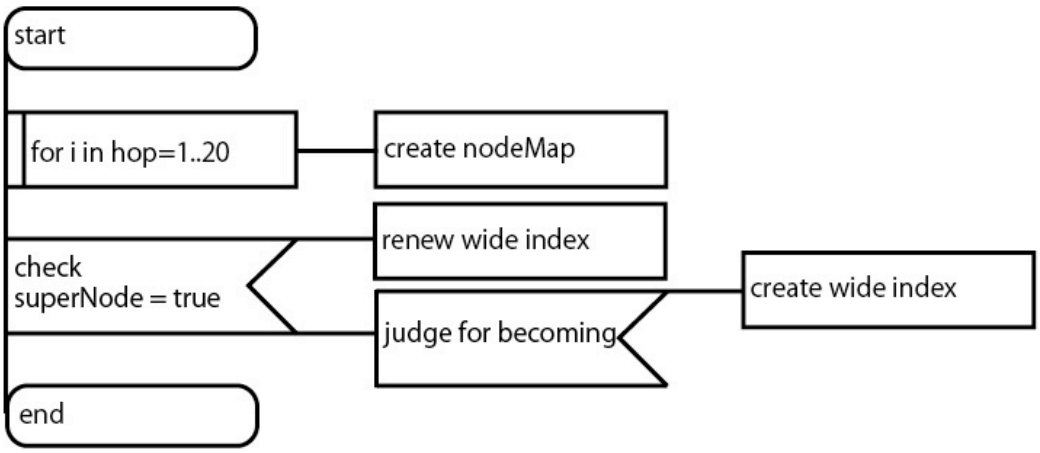

図 4.3 広域インデックス形成および更新処理の全体像

9. 条件を満たす場合のみ，広域インデックスを新規作成する．
10. 広域インデックス更新処理を終了する．

なお，広域インデックスの新規作成及び更新処理を以下に示す．

1. 広域インデックスがあるかどうかをチェックする．
2. ない場合は，インデックス初期化を行う．
3. データベースのノードマップテーブルに登録のある一番ホップ数の大きいノードを 10 件取得する．
4. 取得したノード 10 件を広域インデックスの設定ファイルに登録する．
5. インデックス作成を実行する（ノードからリンクをたどってクロールし続ける）．
6. 100 時間が経過した時点でインデックス作成を終了する．

4.4. 検索処理

本節では，検索に関わる処理の説明を行う．まず，検索実行に必要な機能を説明し，その後，その機能の組み合わせでそれぞれ検索元ノード，ホップ先ノード，スーパーノードでの検索処理の詳細を説明する．

4.4.1 各機能

- ループ防止機能
  本システムは，クエリを順次様々なノードに送信して行くことで検索結果を収集している．そのため，同じクエリが何度も同一のノードにホップされ，クエリがループしてしまうことを防ぐ機能を持たせる．具体的にはユーザがクエリを検索元ノードに入力した際，クエリ ID を自動生成し，クエリの中に含め隣接ノードに送信する．クエリ ID を受け取ったノードは自身が以前受け取ったクエリ ID と同一かどうかをチェックし，同一でない場合にのみ検索を実行する．同一の場合は，検索ストップの旨を応答する．

- ホップ数カウンタ機能
  あるノードが隣接ノードリストに従い，別のノードに対してクエリを送信することをクエリをホップさせるという．クエリを送信しない場合をホップ数 0 とし，隣接ノードに対してのみホップする場合をホップ数 1，その先のノードに対してホップさせる場合は順にホップ数 2，ホップ数 3 と増えて行く．ユーザは検索実行時，検索元ノードにホップ数を入力する．入力されたホップ数はクエリがホップされるごとに 1 ずつ減算され，ホップ数 0 となった時点で，クエリをホップさせることを停止する．

- クエリ送信機能
  本システムではクエリ送信先ノードのリストであるクエリ送信先リストとそれぞれの送信先ノードに対する点数情報が格納されているデータベーステーブル (隣接ノードリスト）を保持する．検索実行時，このリストの点数

情報に従い，高いものから 5 件のノードに対しクエリを送信し，結果を取得する．

- ローカルインデックス検索機能
  ローカルインデックスに対し，検索実行を行う．
- 隣接ノードインデックス検索機能
  隣接ノードインデックスに対し，検索実行を行う．
- 広域インデックス検索機能
  広域インデックスに対し，検索実行を行う．
- xml 生成機能
  全ての検索結果に関しては xml 形式での出力される．また検索元ノードには検索結果のタイトルと URI，ホップを停止したノードの保持している隣接ノードリスト（クエリ未送信先リスト）が表示される．

4.4.2 検索元ノードでの検索処理

検索元ノードでの検索処理を図 4.4 に示し，以下，流れを説明する．

1. クエリが入力される．(キーワード，ホップ数)
2. クエリ ID を生成する．
3. ローカルインデックスに対し，検索を実行する．
4. 隣接ノードインデックスに対し，検索を実行する．
5. 入力されたホップ数が 0 でないならば，データベーステーブルの隣接ノードリストより 5 件取得しクエリ送信．検索結果を取得する．
6. 入力されたホップ数が 0 ならば，データベーステーブルの隣接ノードリストより 5 件取得し，検索結果とする．

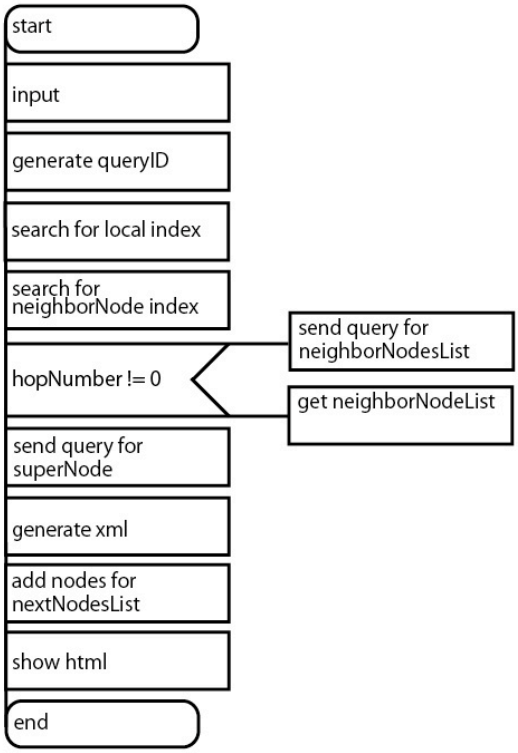

図 4.4 検索元ノードでの検索処理の流れ

7. データベーステーブルのスーパーノードリストより，登録のあるスーパーノードのうち，一番ホップ数の大きいノード 1 件を取得する．

8. 取得したノードに対し，ホップ数 2 でクエリを送信し，検索結果を取得する．

9. 取得した検索結果をすべて合わせ，xml を生成する．

10. 検索結果のうち，クエリ未送信先ノードをデータベーステーブルのノードマップに登録する．

11. html 形式にしてブラウザに表示する．

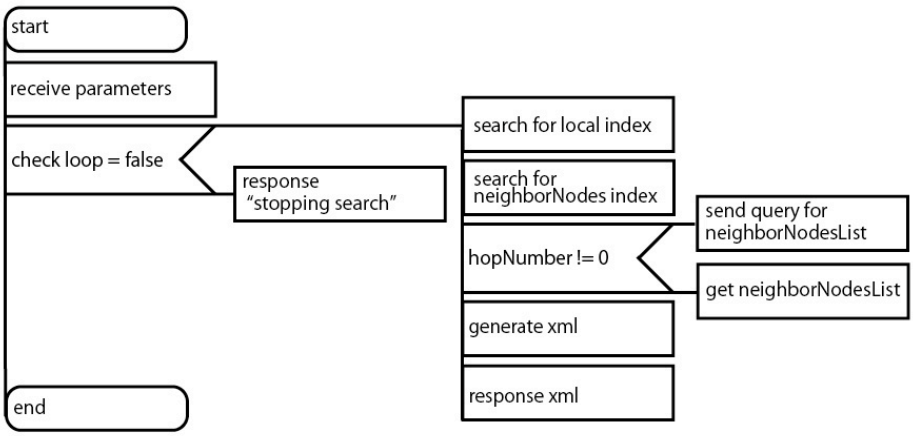

図 4.5 ホップ先ノードでの検索処理の流れ

4.4.3 ホップ先ノードでの検索処理

ホップ先ノードでの検索処理を図 4.5 に示し，以下，流れを説明する．

1. クエリを受け取る．
2. クエリがループしていないかチェックする．
3. ループしているならば検索を停止した旨を応答し処理を終了する．
4. ループしていないならば検索処理を続行する．
5. ローカルインデックスに対し検索を実行する．
6. 隣接ノードインデックスに対し検索を実行する．
7. 受け取ったホップ数が 0 でないならば，データベーステーブルの隣接ノードリストより 10 件取得しクエリ送信．検索結果を取得する．
8. 受け取ったホップ数が 0 ならば，データベーステーブルの隣接ノードリストより 10 件取得し，検索結果とする．
9. 取得した検索結果をすべて合わせ込み，xml を生成する．

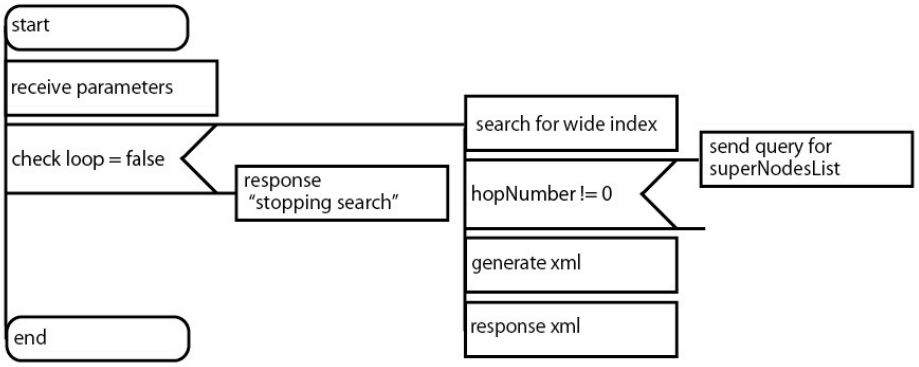

図 4.6 スーパーノードでの検索処理の流れ

10. 生成された xml を応答する.

4.4.4 スーパーノードでの検索処理

スーパーノードでの検索処理を図 4.6 に示し，以下，流れを説明する.

1. クエリを受け取る.
2. クエリがループしていないかチェックする.
3. ループしているならば検索を停止した旨を応答し処理を終了する.
4. ループしていないならば検索処理を続行する.
5. 広域インデックスに対し検索を実行する.
6. 受け取ったホップ数が 0 でないならば，データベーステーブルのスーパーノードリストより 10 件取得しクエリを送信，検索結果を取得する.
7. 取得した検索結果をすべて合わせ込み，xml を生成する.
8. 生成された xml を応答する.

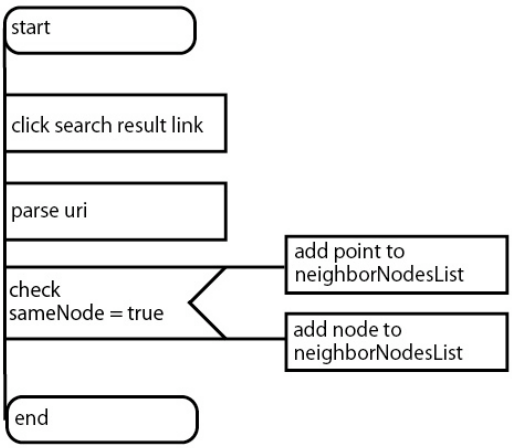

図 4.7 動的経路構成処理の全体像

4.5. 動的経路構成処理

本システムはクエリ送信先リスト (隣接ノードリスト）を保持している．ユーザが検索結果をクリックした際，クリックされた URI を隣接ノードリストに登録し，加点する．すでに登録があった場合は該当ノードのノード点数に加点のみを行う．なお，本研究では，加点を 1 点とするシンプルなアルゴリズムを採用しているが，今後この加点をより高度なアルゴリズムに変更することが可能である．

図 4.7 に処理の流れを示し，以下詳細を説明する．

1. 検索結果を得たユーザが検索結果 URI をクリックする．
2. ノードはクリックされた URI をパースしホスト名とポート番号に分ける．
3. 分けられたホスト名，ポート番号がすでに隣接ノードリストに既に登録があるかどうかを調べる．
4. 登録がない場合は，ホスト名，ポート番号を隣接ノードリストに登録し，ノード点数を 1 点とする．
5. 登録がある場合は，該当するノード情報のノード点数に 1 点を加える．

4.6. まとめ

本章ではあらかじめ実施した予備実験を元に本システムの各処理詳細について設計を行った．インデックス形成及び更新処理では，3 種類のインデックスについて設計を行い，詳細について述べた．検索処理では，検索元ノード，ホップ先ノード，スーパーノードでのそれぞれの検索処理を設計した．動的経路構成処理では，シンプルなアルゴリズムによるノード点数の加点を用いた設計を行った．次章では実際の実装について詳細を述べる．

第5章
実　　装

本章では提案システムの実装について述べる．はじめに提案システムの開発環境を述べる．次に各クラスにおけるメソッドについて述べる．最後にノード情報などを格納しておくデータベースの設計について述べる．

5.1. 開発環境

本システムの開発環境を表5.1に示す．また，使用した全文検索エンジン環境を表5.2に示す．

5.2. インデックス更新クラス

- インデックス設定ファイル更新メソッド
  インデックス対象ノードをデータベースから取得しインデックス設定ファイルに登録する．詳細を図5.1に示す．

- ノードマップ&スーパーノードリスト作成メソッド
  近隣のノードの隣接ノードリストを問い合わせ,ノードマップ及びスーパーノードリストに登録する．詳細を図5.2に示す．

- スーパーノード形成メソッド
  ノードが自律的にスーパーノードとして機能するかどうかを判定する．詳細を図5.3に示す．

表 5.1 開発環境

項目	環境
開発言語	ruby 1.8.7
フレームワーク	RubyOnRails2.3.2
データベース	Mysql1.2.7

表 5.2 全文検索エンジン環境

項目	環境
検索エンジン	HyperEstraier1.4.13
文字コード変換	libiconv1.9.1
可逆データ圧縮	zlib1.2.1
組み込み用データベース	QDBM1.8.75

- 生成される xml の構造
  詳細を図 5.4 に示す.

5.3. 検索クラス

- クエリループ防止メソッド
  クエリ ID を発行し，クエリループを防止する．詳細を図 5.5 に示す.
- クエリ送信メソッド
  ホップ数に応じてクエリを隣接ノードに送信する．詳細を図 5.6 に示す.
- インデックス検索メソッド
  各インデックスに対し，検索を実行する．詳細を図 5.7 に示す.
- 生成される xml の構造
  検索実行により得られた検索結果をすべて xml 形式でまとめる．詳細を図 5.8 に示す.

```
createIndexConfig()
    if indexConfig = nill
        initialize index
    end
    if wideIndex
        get nodesListArray(:10,condition=>hop = max,nodeMap)
    else
        get nodesListArray(845,condition=>point,neighborNodesList)
    end
    open indexConfigFile
    overWrite nodesListArray
    close indexConfigFile
```

図 5.1 インデックス設定ファイル更新メソッド

5.4. 動的経路構成クラス

- 隣接ノード登録メソッド
  ユーザがクリックした検索結果の URI を隣接ノードリストに登録する．詳細を図 5.9 に示す．

5.5. データベース設計

クエリの送信先ノード情報などをデータベースに格納する．データベースは 5 つのテーブルを保持している．以下に各データベーステーブルの詳細を記す．

表 5.3 に自ノード情報を格納するテーブルを示す．”supernode”は，自ノードがスーパーノードとして機能しているかどうかのフラグである．スーパーノードの場合は 1，そうでない場合は 0 を格納する．”name”は自ノードのホスト名を格納する．”port”は自ノードのポート番号を格納する．

表 5.4 にクエリ ID を格納するテーブルを示す．”random”は検索実行時に生成された，もしくはクエリとして受信したクエリ ID を格納する．

表 5.5 に隣接ノードのノード情報を格納するテーブルを示す．”name”は隣接ノー

```
create NodeMapAndSuperNodesList()
    for i in hop = 1..20
        delete nodeMap(:all,:conditions = >hop = i+1)
        get parentNodeArray(:all,:conditions => hop = i )
        parentNodeArray.each{
            HTTPPOSTrequest()
            get Response(neighborNodes.xml)
            element(neighborInfo).each{
                if attribute(superNode) = 1 then
                    find or add element(neighborName,neighborPort)
                    (superNodesList)
                end
                add  element(
                parentName,parentPort,hop=i,neighborName,neighborPort)
                (nodeMap)
            }
        }
    end
```

図 5.2 ノードマップ&スーパーノードリスト作成メソッド

ドのホスト名を格納する．”port”は隣接ノードのポート番号を格納する．”point”はノードの点数を格納する．

表 5.6 にノードマップのノード情報を格納するテーブルを示す．”parentName”は起点となるノードのホスト名を格納する．”parentPort”は起点となるノードのポート番号を格納する．”hop”は，起点となるノードの自ノードからのホップ数を格納する．”childName”は，起点となるノードから見た隣接ノードのホスト名を格納する．”childPort”は，起点となるノードから見た隣接ノードのポート番号を格納する．

表 5.7 にスーパーノードのノード情報を格納するテーブルを示す．”name”はスーパーノードのホスト名を格納する．”port”はスーパーノードのポート番号を格納する．

```
judgeBecomingSuperNode()
    if averageCpuUsage < 50% &&
        averageCpuUsage < 40% &&
        restHDD > 1200GB then
        flag superNode = 1(selfInfo)
    end
```

図 5.3 スーパーノード形成メソッド

```
<GatherNodes>
    <responseNodesList parentName,parentPort,hop>
        <child>
            childName
            childPort
        </child>
    </responseNodesList>
</GatherNodes>
```

図 5.4 インデックス更新クラスで生成される xml の構造

```
notLoopQuery(int queryID)
    if node = searchRootNode then
        queryID = randomNumber(0..100000000000)
        add queryID in QueryIDList
    else
        recieve queryID
        if same queryID = nill in QueryIDList
            add queryID in QueryIDList
            proceed next method
        else
            stop search
        end
     end
```

図 5.5 クエリループ防止メソッド

```
sendQuery(integer queryID, string keyword, integer hopNumber, int hopFirst)
    if hopNumber != 0
        newHopNumber = hopNumber -1
        if node = rootNode
            hopFirst = hopNumber
        else
            hopFirst = hopFirst
        end
        if node= rootNode
            get nodesListArray(:5,condition=>point,neighborNodesList)
            get nodesListArray(:1,superNodesList)
            nodesListArray.each{
               HTTPPOSTrequest(queryID,keyword,hopFirst,newHopNumber)
            }
        else if node = hopNode
            get nodesListArray(:5,condition=>point,neighborNodesList)
            nodesListArray.each{
            HTTPPOSTrequest(queryID,keyword,hopFirst,newHopNumber)
            }
        else if node = superNode
            get nodesListArray(:10,superNodesList)
            nodesListArray.each{
            HTTPPOSTrequest(queryID,keyword,hopFirst,newHopNumber)
            }
        end
        getResponse(resultNeigborSearch.xml,resultSuperSearch.xml)
    else
        if node = superNode
            stop search
        else
            get nodesListArray(:5,condition=>point,neighborNodesList)
        end
    end
```

図 5.6 クエリ送信メソッド

```
searchIndex(string searchKeyword)
    if node != superNode then
        search(searchKeyword) for localIndex
        search(searchKeyword) for neighborNodesIndex
    else
        search(searchKeyword) for wideIndex
    end
```

図 5.7 インデックス検索メソッド

```
<GatherResult>
    <localAndNeighborResult name,port,hop>
        <searchResult>
            title
            uri
        </searchResult>
    </localAndNeighborResult>
    <superNodeResult>
    <wideResult>
            title
            uri
    </wideResult>
    </superNodeResult>
    <nextNode parentName,parentPort,hop>
        <child>
            childName
            childPort
        </child>
    </nextNode>
</GatherResult>
```

図 5.8 検索クラスで生成される xml の構造

```
addNode()
    when userClick result then
        parse uri(name,port)
        find uri(:1,uri = name && port,neighborNodesList)
        if uri = true then
            add point(1)(neighborNodesList)
        else
            add uri(name,port,point = 1)(neighborNodesList)
        end
    end
```

図 5.9 隣接ノード登録

表 5.3 自ノード情報

項目	データ型	説明
supernode	integer	スーパーノードフラグ
name	varchar(256)	ホスト名
port	integer	ポート番号

表 5.4 クエリ ID 管理

項目	データ型	説明
random	integer	クエリ ID

表 5.5 隣接ノードリスト

項目	データ型	説明
name	varchar(256)	ホスト名
port	integer	ポート番号
point	integer	点数

表 5.6 ノードマップ

項目	データ型	説明
parentName	varchar(256)	ホスト名
parentPort	integer	ポート番号
hop	integer	ホップ数
childName	varchar(256)	ホスト名
childPort	integer	ポート番号

表 5.7 スーパーノードリスト

項目	データ型	説明
name	varchar(256)	ホスト名
port	integer	ポート番号

第6章
評　　価

本章では提案したシステムが情報検索システムとしての要求を満たすための基盤になり得ているかを確認するため，各検索手法ごとの検索実行時の情報の網羅性および検索応答時間の比較を行う実験，各検索手法ごとの検索トラフィックの算出，そして検索試行前後の検索ヒット数の比較を行う実験で評価する．

3章で述べたように，ピュアP2Pモデル及びスーパー・ノード・モデルを採用していることでノードのスケーラビリティを維持できているとする．その前提において実験を行う．

検索実行1回あたりに得られる検索対象ノードが増えれば増えるほど情報の網羅性は高くなる．しかし，各検索方法によって検索応答時間が異なるため，それらの違いを明らかにし，対象ノードが増加しても検索応答時間短縮が実現できていることを確認する．さらに各検索手法によって発生する検索トラフィックを明らかにすることで，どういう状況にどの検索手法が有効かを次章で考察する．また，シンプルなアルゴリズムを用いて動的経路構成および優先度に基づくクエリ送信が検索効率の向上に有効なことを確認する．

6.1. 実験環境

実験は表6.1に示す環境の計算機を2台用意し，提案システムを複数プロセス起動させて行った．また，ネットワークは図6.1に示すローカル環境とした．

表 6.1 実験に用いた計算機の環境

CPU	Intel(R) Core(TM)2 Duo E7200（2.53GHz）
Memory	2GB
OS	Ubuntu 8.10
WebServer	apache2.2.9

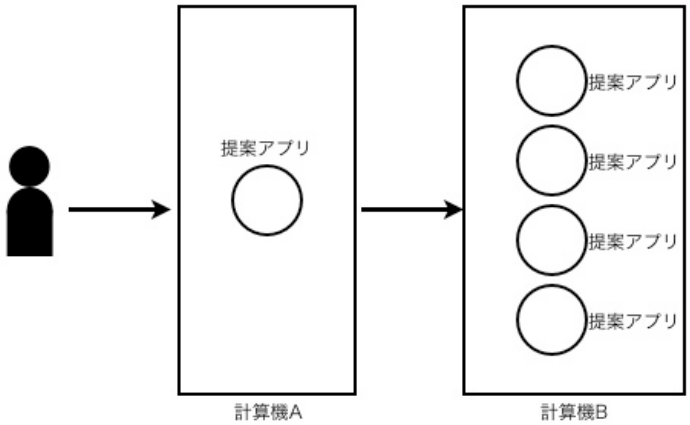

図 6.1 実験環境

6.2. 実験内容

本節では前節で述べた実験環境のもとで行う評価実験の内容について述べる．

6.2.1 各検索手法における検索応答時間と情報の網羅性の比較実験

提案手法では，単純なピュア P2P モデルを元に検索応答時間と情報の網羅性の改善の為に，隣接ノードインデックスと広域インデックスを導入した．様々なトポロジーモデルについて対象ノード数と検索応答時間との関係を明らかにするとともに，隣接ノードインデックスと広域インデックスの有効性を確認する．実験で用いたトポロジーモデルを図 6.2 に示す．

- modelA:ピュア P2P の直列モデル
- modelB：ピュア P2P でクエリを 2 ノードに送信するモデル
- modelC：ピュア P2P でクエリを 3 ノードに送信するモデル
- modelD：隣接ノードインデックスモデル
- modelE：広域インデックスモデル

なお，各ノードに配するコンテンツは約 1MB で全て同一のものとした．

6.2.2 各検索手法におけるトラフィック量の算出

提案で導入したスーパーノードの広域インデックスへの検索，一般ノードの保持するローカルインデックスおよび隣接ノードインデックスへの検索の各手法においてそれぞれクエリの応答トラフィック量を明らかにする．

算出のために，まず 1 ノードから複数ノードへの検索を行った際のクエリの応答トラフィック量を測定し，それらのデータを元にさらに多くのノードへ検索を行った際の応答クエリのトラフィック量を算出する．各算出対象として，以下を対象とした．

- 一般ノード
  ローカルインデックスと隣接ノードインデックスを保持する一般のノードに対し 5 ノードずつクエリ送信していく場合を想定した．
- スーパーノード
  前章で設計した通り，一般ノードから 1 ホップでスーパーノード 1 台にクエリ送信が行われ，2 ホップ目でさらに 10 台のスーパーノードへクエリ送信する場合を想定した．
- 両方（一般ノードとスーパーノード）
  上記 2 つの場合を足し合わせた場合を想定した．なお，3 ホップ以上になってもスーパーノードは 2 ホップまでとし，一般ノードが増加していくこととする．

なお，各ノードに配するコンテンツは約 1MB で一般ノードおよびスーパーノード全て同一のものとして算出した．

6.2.3 検索試行前後における検索効率の比較実験

提案手法では，検索される頻度の高いキーワードに対し検索効率を向上させるため動的経路構成と優先度に基づくクエリ送信機能を導入した．ユーザは，検索元ノードから検索を行う際，検索元ノードが保持しているコンテンツに近いクエリを入力する場合が多いと考えられる．そのため，十分な回数の検索試行を行った後は類似コンテンツを保持したノードがより少ないホップ数になると考えられる．簡単なトポロジーモデルを用いて，十分な回数の検索試行前後でホップ数ごとの検索ヒット数の比較をする．

トポロジーモデルの検討

本実験で用いるトポロジーモデルについて考察を行う．本実験で用いるトポロジーモデルは現実の運用に則して以下の要件を満たしている必要がある．

- 各ノードには必ず次の送信先ノードが存在する．
- ループする部分がある．

また実験の初期状態は，どのノードにとっても条件が均一になるようにするために以下の要件を満たすこととする．

- 初期状態は、全てのノードから全てのノードにクエリが届くように配置し情報の網羅性を 100%とする．
- 全てのノードは 2 種類のコンテンツのどちらか一方を保持し全体としてその割合は 50%ずつとする．
- 初期状態は全てのノードから見て均等にその 2 種類のコンテンツが配されるようにする．

採りうるトポロジーモデルを 6.3 に示し，それぞれが上記の条件を満たすかどうかの考察を行う．

- modelF
  全てのノードがランダムに配され，ランダムにループしている点で一番現実のトポロジーに近い．しかしながら実験の初期状態としては全てのノードから全てのノードにクエリが届かない場合もあり，また、全てのノードから見て均等に 2 種類のコンテンツが配されていない．

- modelG
  ノード a やノード b から見たツリー構造である．ノード a やノード b から見ればコンテンツは均等に配置することができるものの，4 ホップ目つまりノード c などではクエリの送信先がない．全てのノードから全てのノードにクエリが届かない構造である．

- modelH
  リング構造である．これは上記の条件を全て満たす．

- modelI
  リング構造で，modelH に加えクエリ送信先を各ノード 2 つとし一方のクエリ送信先がもう一方のクエリ送信先の 3 ホップ先となるようにされている．これも上記の条件を全て満たす．また，modelH に比べ全てのノードに検索実行をし検索結果を得る際のホップ数が半分である．

これらの考察から modelH および modelI が今回の実験に用いるトポロジーモデルの候補であるが，今回は検索試行前後で各ホップ数での検索ヒット数の差を比較したいためよりホップ数の多い modelH を採用するものとする．

実験内容

ノード数 20 台でそれぞれ 20 回，合計 400 回の検索試行を行い，検索前後でホップ数ごとの検索ヒット数を比較する．同一ホップ数での検索ヒット数が増加する

ことを確認することで，検索効率が向上していることが確認できる．実験の初期トポロジーを図 6.4 に示す．

検索キーワード「ruby」にヒットするコンテンツを持つノード 10 台，検索キーワード「kmd」にヒットするコンテンツを持つノード 10 台を交互に接続し，リング状とした．

なお，検索試行の前後でノード a においてそれぞれのホップ数ごとの検索ヒット数を測定した．また，検索試行後にノード a において「ruby」と入力した際のノードマップ，ノード A において「kmd」と入力した際のノードマップを作成した．

6.3. 実験方法

6.3.1 各検索手法における検索応答時間と情報の網羅性の比較実験

検索キーワード「kmd」とそれぞれのホップ数を入力し，検索実行ボタンをクリックした時から，検索結果が表示されるまでの時間を測定した．またその際に，表示された検索結果のヒット数を数えた．各ノードに配しているコンテンツは同一のものであるため，検索結果のヒット数は，クエリを送信したノード数と同一になる．また，キャッシュの関係を排するため，検索実行時には全て apache を再起動させた．

6.3.2 各検索手法におけるトラフィック量の算出

検索キーワード「kmd」とそれぞれのホップ数を入力し，検索実行ボタンをクリックした時から検索結果が表示されるまでを 1 サイクルとし，1 サイクルが修了すると自動で再検索を行うプログラムを作成し実行した．2 分間プログラムを稼働させ，その間に測定された合計トラフィック量を元に 1 秒あたりの平均トラフィック量を算出した．

6.3.3 検索試行前後における検索効率の比較実験

検索試行の際，検索キーワード「ruby」にヒットするコンテンツを持つノードには「ruby」を，検索キーワード「kmd」にヒットするコンテンツを持つノードには「kmd」を検索キーワードとして入力し，その都度ヒットした検索結果をランダムにクリックし，隣接ノードリストに登録していくことにした．なお，ホップ数はすべて 20 とした．

6.4. 実験結果

本節では前節で述べた実験の結果を示す．

6.4.1 各検索手法における検索応答時間と情報の網羅性の比較実験

各検索手法における検索応答時間と情報の網羅性の比較実験を図 6.5 に示す．modelD を除いて，すべてのモデルにおいて対象ノード数が増加するのに比例して検索応答時間が増加している．しかしながら，すべてのモデルで同様の増加率ではなく，モデルによって大きく変わっている．modelA は全てのモデルの中で一番高い増加率である．modelB と modelC はほぼ同一の増加率である．なお，modelB はホップ数 6 で，modelC はホップ数 5 でそれぞれタイムアウトエラーとなった．そのため，modelB は 63 台，modelC は 40 台までの検索応答時間しか計測できなかった．modelD は一番増加率が低い．modelE は modelE よりも増加率は高いが，modelB 及び modelC よりも低い．

6.4.2 各検索手法におけるトラフィック量の算出

各検索手法におけるトラフィック量を図 6.6 に示す．

算出した結果，5 ホップで一般ノードへの検索時のクエリの応答トラフィックは 1 秒あたり 14.79MB となった．一方スーパーノード群への検索時には対象ノー

ド数が最大 11 台であるため 2 ホップで応答トラフィックは 1 秒あたり 0.04MB となった．

6.4.3 検索試行前後における検索効率の比較実験

検索試行前後における検索効率の比較実験を図 6.7 に示す．

検索試行前は検索キーワード「ruby」「kmd」ともに，ホップ数の増加とともに検索ヒット数が増加した．また，ホップ数を 20 にした場合，どちらのキーワードも検索ヒット数が 10 となり，すべてのノードに対し，検索を実行している．一方，検索試行後は検索キーワード「ruby」に関しては 1 ホップで検索ヒット数は 3，2 ホップでは 4，3 ホップでは 6，4 ホップでは 8 となり，いづれも検索施行前よりも検索ヒット数が多い．しかしながらホップ数 5 以上に設定を行っても，クエリループ防止機能が働きそれ以上検索ヒット数が増加することはなかった．また，検索試行後の検索キーワード「kmd」に関してはいづれのホップ数でも 1 件もヒットしなかった．

なお，検索試行時のノード a から見たノードマップの推移を図 6.8 に示す．検索試行後のノードマップでは，5 ホップ以降はループ防止機能が働くためそれ以上ノードが存在しないこととなった．ノードマップに現れたノードはすべて検索キーワード「ruby」にヒットするコンテンツを持つノードであった．しかしながら検索キーワード「ruby」にヒットするコンテンツを持つノードでも 2 ノードが現れなかった．検索キーワード「kmd」にヒットするコンテンツを持つノードは 1 ノードも現れなかった．

また，検索試行終了後のノード A から見たノードマップを図 6.9 に示す．

検索試行後のノードマップでは，6 ホップ以降はループ防止機能が働くためそれ以上ノードが存在しないこととなった．ノードマップに現れたノードはすべて検索キーワード「kmd」にヒットするコンテンツを持つノードであった．しかしながら検索キーワード「kmd」にヒットするコンテンツを持つノードでも 1 ノードが現れなかった．検索キーワード「ruby」にヒットするコンテンツを持つノードは 1 ノードも現れなかった．

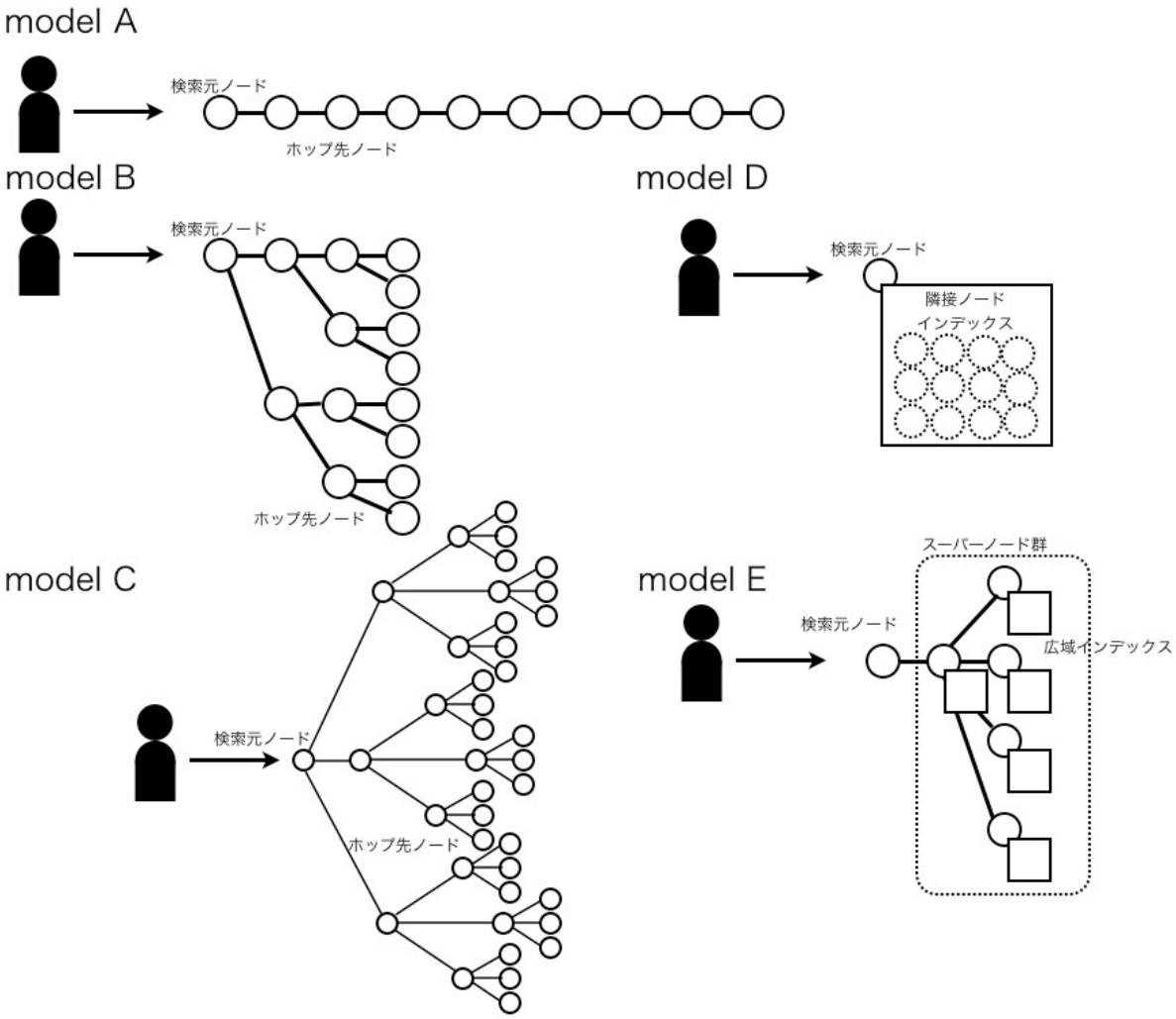

図 6.2 利用したトポロジーモデル

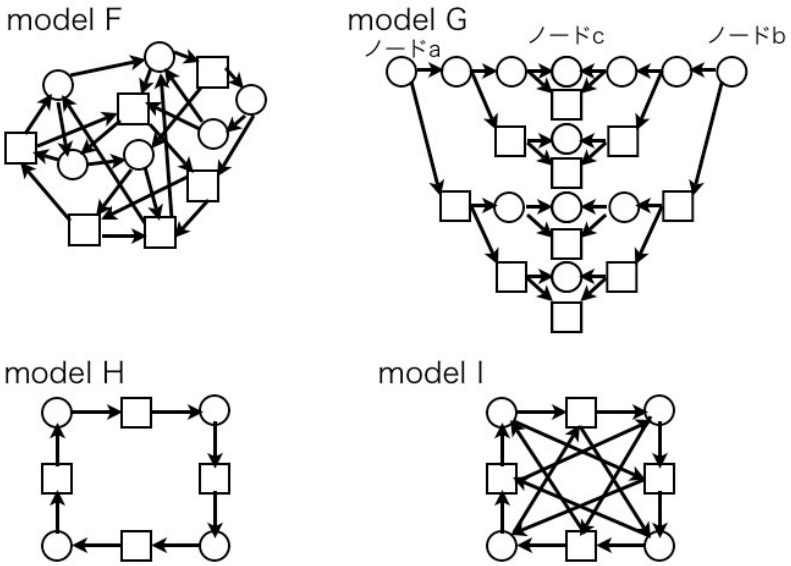

図 6.3 検討するトポロジーモデル

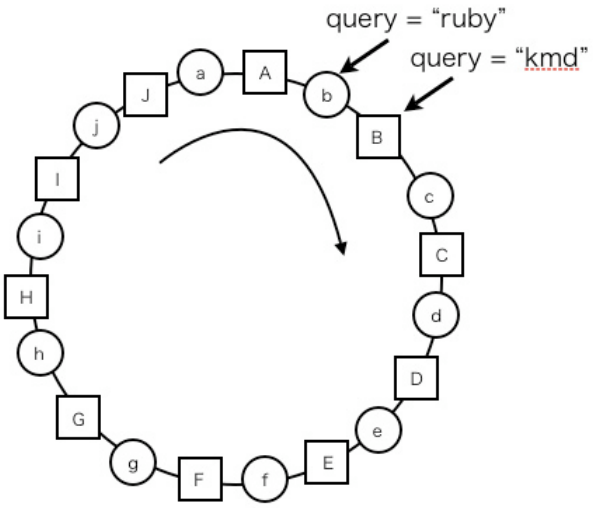

図 6.4 初期トポロジー

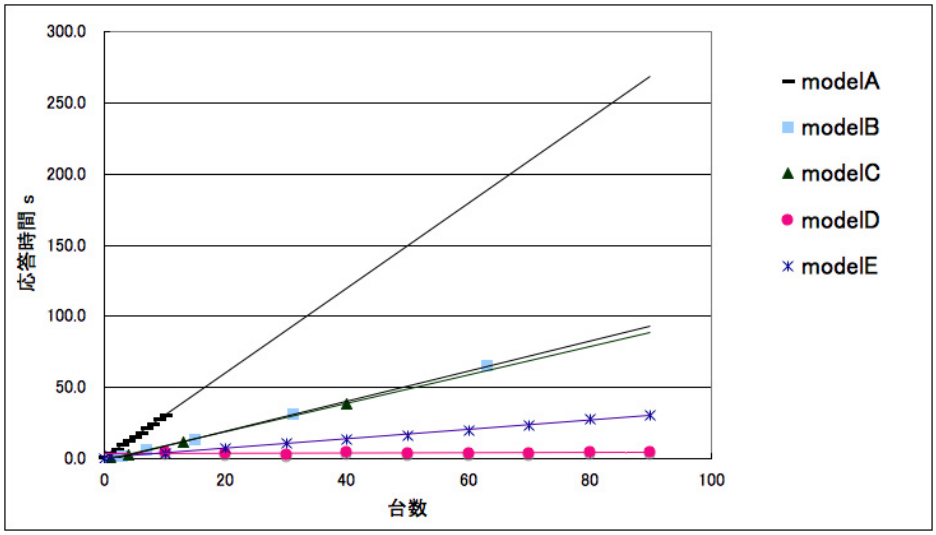

図 6.5 検索対象ノード数と検索応答時間

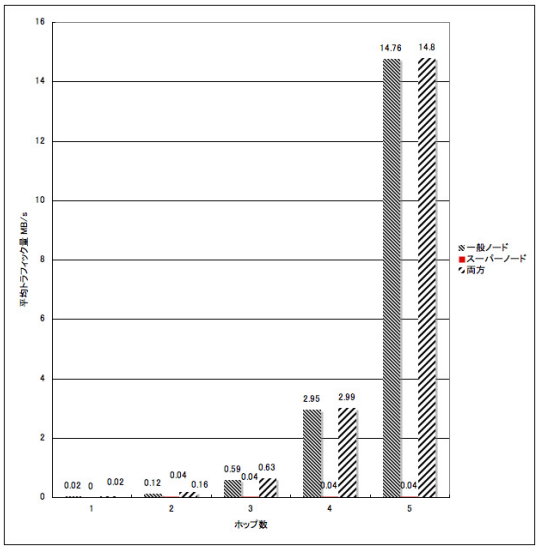

図 6.6 各検索手法における平均トラフィック量

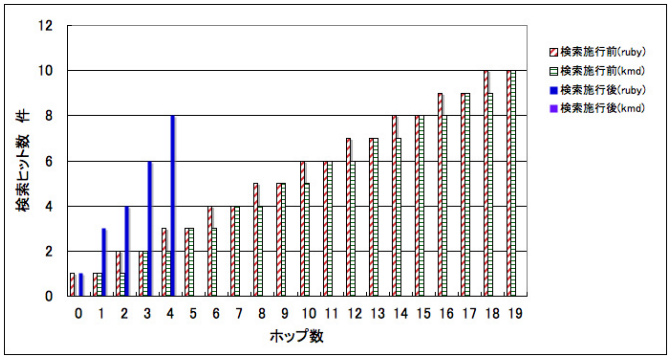

図 6.7 各ホップ数における検索ヒット数の変化

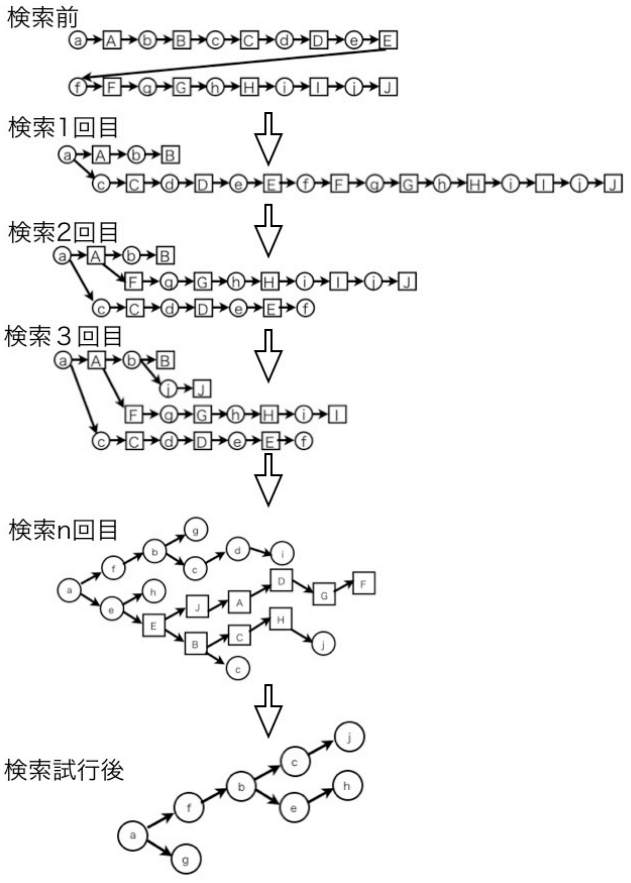

図 6.8 ノード a から見たノードマップ推移

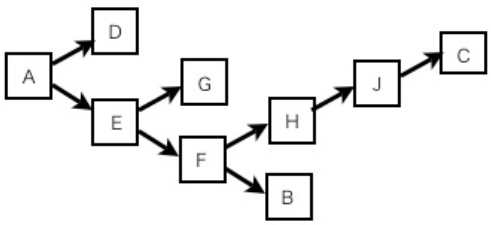

図 6.9 ノード A から見たノードマップ

第7章
考　　察

前章の実験結果を元に本章では各検索手法における検索応答時間と情報の網羅性の比較実験および検索試行前後における検索効率の比較実験について考察する．

7.1. 各検索手法における検索応答時間と情報の網羅性の比較実験に対する考察

modelA は単純なピュア P2P の直列モデルであるため，当初の懸念通り台数を増やすことがつまりホップ数を増やすことになり検索応答時間がかかってしまった．

同じピュア P2P のモデルでも，クエリを複数ノードに送信した modelB 及び modelC は modelA に比べて検索応答時間を短縮することができている．そのため，ある一定のノード数に対し並列してクエリを送信することは検索応答時間短縮に効果があると言える．しかしながら，クエリ送信先数の違う modelB と modelC がほぼ同じ増加率であるのは，ホップ数の減少による検索応答時間の減少と，並列してクエリ送信する数の増加に伴う検索応答時間の増加が相殺された結果であると考える．なお，実験で modelB がホップ数 6，modelC がホップ数 5 でタイムアウトエラーになってしまったことから，クエリフラッディングによる手法では多くのノードを検索対象とすることができないことが判明した．

また，modelD は隣接ノードインデックスを想定したモデルであるが，ホップ数が 0 であるため一番検索応答時間が短く，短い検索応答時間でより多くのノード数を対象に検索するのに有効であると考える．

さらに，modelE は広域インデックスを想定したモデルであるが，必ず検索元

ノードからスーパーノードにクエリを送信しさらにスーパーノード群へもクエリを送信するため，そのホップ数分だけ，modelD よりも検索応答時間がかかってしまう．ただし，広域のインデックスを保持しているノードへの検索であるために同一ホップ数であれば modelB や modelC よりも遥かに多くのノード数を対象とすることができ，本システムにとって情報の網羅性を確保するのに有効であると考える．

以上のことより隣接ノードインデックスが検索応答時間の短縮に，広域インデックスが情報の網羅性の確保に有効であることが確認できた．

7.2. 各検索手法におけるトラフィック量に対する考察

一般ノードへのクエリフラッディングした際，算出したデータから 3 ホップで 1 秒あたり 0.59MB，4 ホップで 1 秒あたり 2.95MB，5 ホップで 1 秒あたり 14.76MB のクエリの応答トラフィックが発生していることが判明した．ネットワークへの負荷を考慮すると 5 ホップした場合はかなりその負荷が大きい．環境にもよるが 4 ホップまでが実用に耐えうるものであると考える．また，今後より多くのホップ数でも実用に耐えうるようにトラフィック量を軽減する仕組みを検討する必要がある．一方，スーパーノードへの検索時には 2 ホップで 11 台のスーパーノードを対象とし 1 秒あたり 0.04MB であることから一般ノードに比べより少ない応答トラフィックでより多くの情報に対して検索が可能であることが判明した．

7.3. 検索試行前後における検索効率の比較実験に対する考察

実験結果より，検索キーワード「ruby」にヒットするコンテンツを持つノードからの検索キーワード「ruby」で検索実行した際，より少ないホップ数で検索ヒット数が増加していることから動的経路構成および優先度に基づくクエリ送信の機能がより検索効率を向上させていることを確認できた．

なお，今回の評価では検索結果をクリックした際に隣接ノードに登録し，そのノードに対し点数を１点追加するという非常にシンプルなアルゴリズムで実施したが，検索効率を向上することができた．今後，より高度なアルゴリズムをプラグインとして投入することでさらに検索効率を向上することが可能であると考える．

7.4. スーパーノードと一般ノードの存在意義に対する考察

全ての実験結果を勘案することによってスーパーノードと一般ノードの存在意義に対して考察を行う．スーパーノードへの検索時には先の評価から検索応答時間や応答トラフィックの検索コストが非常に低く済むことが判明した．一方，一般ノードへの検索コストは非常に高くなることが判明した．しかしながら動的経路構成の効果によって検索頻度の高いキーワードに対する検索結果を得られやすいことも判明した．つまりそれぞれにメリットデメリットが存在しており，これらはトレードオフの関係である．そのため様々な検索ニーズに応えるためには両方をある割合で活用する必要があることがわかった．そのバランスについては今後より実運用に近い環境での評価を通じて検討する必要があると考える．

7.5. まとめ

各検索手法における検索応答時間と情報の網羅性の比較実験から隣接ノードインデックスが検索応答時間の短縮に，広域インデックスが情報の網羅性の確保に有効であることが確認できた．クエリフラッディングによる手法はホップ数が増えると検索応答時間時間がかかってしまいタイムアウトエラーになってしまう．

各検索手法におけるトラフィック量の算出から，ホップ数が増えるに従いネットワーク負荷が高くなることが確認できた．またスーパーノード群への検索は検索コストを低く抑えることが確認できた．

検索試行前後における検索効率の比較実験から動的経路構成および優先度に基づくクエリ送信の機能が検索効率の向上に対し有効であることが確認できた．

さらにスーパーノードへの検索，一般ノードへの検索はそれぞれメリットデメリットがあり，そのバランスについては今後検討する必要がある．

そのため隣接ノードインデックス，広域インデックス，動的経路構成および優先度に基づくクエリ送信の全てを実装することが必要でそうすることで検索応答時間の短縮および情報の網羅性の確保をしつつ，さらに検索効率を向上させられるものであると言える．

第8章
今後の課題

8.1. システムの改善

本研究で提案したシステムではまだ実装できていない機能がある．また，評価をすることで新たに必要となった技術的検討事項がある．それらを以下に述べる．

- 初期トポロジーの検討
  提案システムが実際に稼働を始める際，どのようなトポロジーで開始するべきかの検討が必要である．初期状態としては全てのノードから全てのノードにクエリが届く必要があり，リング型がその一つの候補だと考える．

- 新規ノードの参加方法
  提案システムではノードが参加した後の状態を前提としていた．そのため新規ノードの参加方法は今後の重要な課題である．後述する通り今後 apache モジュールとして展開する．このモジュールをダウンロードサイトからダウンロードする際，自動で隣接ノードリストに一定数のノードをランダムに登録するようにすれば，クエリの送信先は確保できる．またランダムに登録する際，最近ダウンロードしたばかりのノードを登録する比重を高くすれば，本システムを利用し始めてから早い段階で他ノードからも検索されるようになる．また，スーパーノードによる広域インデックスではサイトのリンクをたどってクローリングするため，そちらからも検索されるようになる．

- 離脱ノードがあった場合の扱い
  提案システムでは離脱ノードがあった場合の扱いについては言及しなかっ

た．しかしながら実際の運用では，サイトが閉鎖したりメンテナンスのためにノードが離脱することは日常的に発生する．そこで送信先のノードからの応答がなかった場合，ノードマップに従ってその先のノードにクエリを送信し直す機能を追加する．また，一定時間応答がない場合は，隣接ノードリストから削除する機能も必要である．

- クエリ未送信先結果表示機能の活用
  提案システムでは情報検索時にホップ数が 0 となったノードではそれ以上クエリを送信せずに，隣接ノード情報のみを自ノードの検索結果とともに応答する．この情報はノードマップ作成時に活用されるが，さらに活用できるものと考える．例えば一度情報検索を行ったものの欲しい情報が得られなかった場合の再検索開始ノードとして活用できると考える．

- 情報のマネージャビリティの向上
  提案システムは集中インデックス型に比べ情報のマネージャビリティは確保しやすい．しかしながら隣接ノードインデックス，スーパーノードによる広域インデックスを用いているため，ピュア P2P モデルよりは確保しづらい．今後はあるノードで情報が削除された際，各インデックスが連動して更新される仕組みが必要であると考える．

- 重複した情報の扱い
  各ノードがそれぞれにローカルインデックス，隣接ノードリストインデックス，広域インデックスの 3 種類のインデックスに情報検索を行うため，検索結果が重複する可能性が高い．重複した検索結果をそのままユーザに検索結果として表示すると非常に見づらい．そのため重複した検索結果は検索元ノードでの検索結果表示時に自動でまとめることで重複して表示されない仕組みが必要である．またさらに重複した検索結果はそれだけ検索クエリにマッチしてるとも言えるため，ランキングアルゴリズムに活かすことも可能であると考える．

- クエリフラッディングのホップ数増加
  評価結果から，2 ノードずつクエリ送信していくモデルでも 6 ホップでタイ

ムアウトになってしまうことが判明した．情報の網羅性に関しては広域インデックスを用いることで確保できるものの，クエリフラッディングの手法においても性能向上が必要である．そのためにはソースコードのチューニングや apache のチューニングが必要である．また，現在の仕様では全ての検索対象ノードからの応答を受け取ってから検索結果を表示しているが，各ノードで他ノードからの応答がなくても自ノードのみの検索結果表示し応答があり次第，追加で応答を出すような非同期処理の仕組みを入れ込むことも有効であると考える．

- スーパーノードと一般ノードの割合
  前章でも考察した通り，本提案システムはスーパーノードと一般ノードの両方が必要でそれらがある割合で存在している必要がある．しかしながらその割合については今後より実運用に近い環境での評価を通じて検討する必要がある．

- プライバシー問題
  本システムはユーザが各 web サーバにアクセスをしてそこで検索クエリを入力することを前提としているため，本来ならば各ノードへは様々なユーザのクエリが入力され，動的経路構成では嗜好性の近い様々なユーザの検索結果が反映される．しかしながら個人でサイトを開設する人も多く，そのような場合検索ユーザはサイトを開設した本人のみといった場合も考えられる．その際の隣接ノードインデックスは個人の嗜好を反映してしまうため，プライバシーが問題になる可能性がある．この点に関しては実際の運用の中でどの程度それが問題になるのかを議論していく必要がある．

8.2. 今後の普及のための方策

本研究で提案した P2P 型の情報検索システムはより多くの web サーバに導入されることで，その有効性は初めて実現させるものである．そのため，普及のた

めの方策が必要である．以下に普及のための方策を述べる．

- apache モジュールとしての展開
  本システムを apache のモジュールとして展開する．広く普及している web サーバである apache のモジュールとして展開することで，様々な web サーバに導入してもらえるものとすることが必要である．そうすることで，今回は小規模な環境での評価にとどまったものの，より大規模な環境での評価も可能となり今回発見できなかった問題点などが発見できる可能性が高い．

- オープンソースコミュニティの立ち上げ
  apache のモジュールとして展開する際に，本システムのソースコードをオープンソースとして公開する．さらにオープンソースコミュニティを運営していく．そうすることで，継続的により多くの開発者からシステムの改善を期待でき，より信頼できる有用なシステムにしていくことが可能である．オープンソースコミュニティの運営を通じて情報検索というインターネットを利用する際に必要不可欠な基盤をより多くの人に活用してもらえるようにすることが今後の課題である．

第9章
結　　論

増え続けるインターネット上の情報を持続的に検索できるシステムの必要性からP2Pネットワークを基本にした情報検索システムの提案を行った．本提案システムを用いることで，インターネット上の情報を全て一カ所に集めインデックス化することがないため，今後も増え続ける情報に対し持続的に検索サービスを提供し続けることが可能となる．本提案システムはP2Pネットワーク技術を用いて個々のノードのスケーラビリティや情報のマネージャビリティを確保している一方で，隣接ノードインデックスや広域インデックスを導入することで，検索応答時間を短縮し，高い情報の網羅性を実現している．また，動的経路構成および優先度に基づくクエリ送信により．ユーザが検索する頻度の高いキーワードに対して検索効率を向上させることを実現している．

本提案システムの有用性を確認するために各検索手法における検索応答時間と情報の網羅性の比較実験，トラフィック量の算出，および検索試行前後における検索効率の比較実験を行った．実験の結果，隣接ノードインデックスが検索応答時間の短縮に効果があることが判明した．また，広域インデックスが情報の網羅性の向上に効果があることが判明した．またスーパーノード群への検索は検索コストが非常に抑えられる一方で，一般ノードへの検索は新しいコンテンツや検索頻度の高いキーワードへの検索に有効であるとあることがわかった．さらに動的経路構成と優先度に基づくクエリ送信が，検索効率を向上させるには有効であり，今後，高度なアルゴリズムを投入することでさらに検索効率を上げることが可能ということが確認できた．

今後の課題として，システムの改善の必要がある．初期トポロジーの検討，ノードの参加方法や離脱の扱い，クエリ未送信先結果表示機能の活用，重複した検索

結果の扱いに対する機能追加，情報のマネージャビリティの向上，クエリフラッディング手法のチューニング等である．さらに今後普及させるための方策を検討する必要がある．apache のモジュールとして展開しより多くの web サーバに導入してもらうことができ，P2P システムとしての有効性を高めるとともに，大規模な評価実験を行うことができるようになる．またオープンソースとしてソースコードを公開し，継続的により多くの開発者からシステムの改善を期待できるコミュニティを運営することでより信頼のできる有用なシステムにしていく必要があると考える．

謝　　辞

本研究の主指導教員であり，幅広い知見から的確な指導と暖かい励ましやご指摘をしていただきました慶應義塾大学大学院メディアデザイン研究科の砂原秀樹教授に心から感謝いたします．また本研究の副指導教員であり，絶えずご指導とご助言をいただきました慶應義塾大学大学院メディアデザイン研究科の稲蔭正彦教授に心から感謝いたします．

研究の方向性について様々な助言や指導をいただきました慶應義塾大学大学院メディアデザイン研究科の加藤朗教授に心から感謝いたします．同様に，研究の方向性について様々な助言や指導をいただきました慶應義塾大学大学院 メディアデザイン研究科の杉浦一徳准教授に心から感謝いたします．

研究活動，学生生活全般にわたり，数多くの貴重な助言，ご指導をいただいた，廣海緑里氏，山内正人氏に心より感謝いたします．

さまざまな面から研究活動を支えていただき，時に苦楽を共にした慶應義塾大学大学院メディアデザイン研究科 NetworkMediaProject の皆様に心から感謝いたします．

最後に，研究活動に関する理解とともに，温かく支えてくれた妻と絶えず笑顔を振りまいてくれた娘に心から感謝します．

参 考 文 献

[1] April 2009 Web Server Survey -Netcraft. `http://news.netcraft.com/`.

[2] Google. `http://www.google.co.jp/`.

[3] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, Vol. 37, No. 5, p. 43, 2003.

[4] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI ' 06), 2006.

[5] M. Burrows, et al. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th OSDI*, Vol. 11, 2006.

[6] L.A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *IEEE micro*, Vol. 23, No. 2, pp. 22–28, 2003.

[7] 首藤一幸, 加藤大志, 門林雄基, 土井裕介. 構造化オーバレイにおける反復探索と再帰探索の比較. 情報処理学会研究報告, pp. 103–2, 2006.

[8] H. Chen, H. Jin, J. Wang, L. Chen, Y. Liu, and L.M. Ni. Efficient multi-keyword search over p2p web. 2008.

[9] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 175–186. ACM New York, NY, USA, 2003.

[10] 松波秀和, 寺田努, 西尾章治郎. P2P 型コンテンツ検索システムにおけるコンテンツ分布を考慮した Top-k 検索処理手法. 電子情報通信学会第 17 回データ工学ワークショップ (DEWS 2006) 論文集 (Mar. 2006 to appear).

[11] A. Kumar, J. Xu, and E.W. Zegura. Efficient and scalable query routing for unstructured peer-to-peer networks. In *IEEE INFOCOM*, Vol. 2, p. 1162. Citeseer, 2005.

[12] D. Zeinalipour-Yazti, V. Kalogeraki, and D. Gunopulos. Information Retrieval in Peer-to-Peer Networks. *CISE 2003*.

[13] P2P ネットワークの在り方に関する作業報告部会報告書. `http://www.soumu.go.jp/s-news/2007/pdf/070629_11_1.pdf`.

[14] F. Menczer, L.S. Wu, and R. Akavipat. Intelligent peer networks for collaborative web search. *AI Magazine*, Vol. 29, No. 3, pp. 35–45, 2008.

[15] P. Haase, R. Siebes, and F. Van Harmelen. Peer selection in peer-to-peer networks with semantic topologies. *Lecture notes in computer science*, pp. 108–125, 2004.

[16] 森下広史, 河野浩之. セマンティックな確率的 P2P ルーティングの提案. 第 21 回人工知能学会全国大会 (JSAI2007), 1G1-5, CD-ROM, ISSN, pp. 1347–9881.

[17] E. Cohen, A. Fiat, and H. Kaplan. Associative search in peer to peer networks: Harnessing latent semantics. *Computer Networks*, Vol. 51, No. 8, pp. 1861–1881, 2007.

[18] 福元良太, 荒川伸一, 村田正幸. べき則の性質を有するネットワークにおけるオーバレイによる経路制御手法の性能評価 (トラヒック解析, FMC, モバイルネットワーク, 情報家電ネットワーク及び一般). 電子情報通信学会技術研究報告. IN, 情報ネットワーク, Vol. 106, No. 358, pp. 19–24, 2006.

[19] 吉田紀彦, 内田良隆, 楢崎修二, 瀬川淳一, 下川俊彦. インデクスサーバの自律形成によるピアツーピアシステムの動的効率化. 電子情報通信学会論文誌 B, Vol. 86, No. 8, pp. 1445–1453, 2003.

[20] 全文検索システム HyperEstraier. `http://hyperestraier.sourceforge.net/`.

[21] J. Risson and T. Moors. Survey of research towards robust peer-to-peer networks: Search methods. *Computer Networks*, Vol. 50, No. 17, pp. 3485–3521, 2006.

[22] 江崎浩. P2P 教科書 インプレス標準教科書シリーズ. インプレス R&D, 2008.

[23] 金子勇. Winny の技術. アスキー, 2005.

[24] 西田圭介. Google を支える技術 巨大システムの内側の世界. 技術評論社, 2008.

付録
Alexandria Digital Library インストールマニュアル

A.1. はじめに

本マニュアルは，Alexandria Digital Library システムをサーバにインストールする際の手順をまとめたものである．
Alexandria Digital Library システムは以下のステップにより行われる．

- 基本コンポーネントのインストールおよび設定．
- web 公開のためのインストールおよび設定．
- HyperEstraier1.4.13 と関連モジュールのインストール．
- Alexandria Digital Library コンポーネントのインストールおよび設定．

なお，本マニュアルは全て以下の OS を前提としている．
前提　OS:Ubuntu 8.10
インストールを始める前にユーザ名の確認をする．
$ id
uid = 1000(ユーザ名) と出てくるユーザ名を以下マニュアルの USER に代入のこと．(1000 は環境によって変わる．)

A.2. 基本コンポーネントのインストールおよび設定

基本コンポーネントのインストールおよび設定は以下の手順で行われる．

- Ruby1.8.7 および関連モジュールのインストール.
- Rubygems1.3.1 のインストール.
- Build-essential C(C++) 環境のインストール.
- MySql2.7 のインストール.
- Ruby on Rails2.3.2 のインストール.
- OpenSSL 関連モジュールのインストール.
- SunJavaJDK のインストール.
- libxml2-dev および libxslt-dev のインストール.

以下，インストールを実施する.

1. Ruby1.8.7 のインストール
   apt-get コマンドのアップデートをした後，Ruby および Ruby 関連のインストールを行う.
   $ sudo apt-get update
   $ sudo apt-get install ruby-dev ruby ri rdoc irb libreadline-ruby libruby libopenssl-ruby -y
   Ruby のインストール確認.
   $ ruby -v
   ruby1.8.7(2008-08-11 patchlevel 72)[i486-linux] と表示されることを確認する.
   表示されなければもう一度やり直す.

2. Rubygems1.3.1 のインストール.
   ソースファイルの保存先ディレクトリを作成した後，Rubygems の圧縮ファイルをダウンロードし，解凍するその後インストールし，関連シンボリックリンクを張る最後に Rubygems 関連のアップデートを実行する.
   $ mkdir /home/USER/sources (USER は各自のユーザ名を合わせて入力の

こと．）

$ cd /home/USER/sources

$ wget http://rubyforge.org/frs/download.php/45905/rubygems-1.3.1.tgz

$ tar xvzf rubygems-1.3.1.tgz

$ cd rubygems-1.3.1

$ sudo ruby setup.rb

$ sudo ln -s /usr/bin/gem1.8 /usr/bin/gem

gem に含まれるモジュールのアップデート．

$ sudo gem update

nothing to update と表示されることを確認する．

gem のアップデート．

$ sudo gem update –system

nothing to update と表示されることを確認する．

3. C（C＋＋）環境のインストール．
   build-essential-11.4 のインストール（途中，y / n?と聞かれるので y を入力する）．
   $ sudo apt-get install build-essential
   インストールできていることを確認する．
   $ dpkg -l |grep build-essential
   ii build-essential 11.4 と出てくることを確認する．

4. Mysql2.7 のインストール．
   途中，パスワードを求められるので各自で決めたパスワードを入力する．
   $ sudo apt-get install mysql-server mysql-client libmysqlclient15-dev libmysql-ruby1.8 -y
   $ sudo gem install mysql
   mysql がインストールされているか確認する．
   $ gem list |grep mysql
   mysql(2.7) と表示されることを確認する．

5. Ruby on Rails2.3.2 のインストール.
   Ruby on Rails のインストール後，Ruby on Rails のシンボリックリンクを張る.
   $ sudo gem install rails
   $ sudo ln -s /usr/bin/rails /usr/local/bin/rails
   Ruby onRails がインストールされているか確認する.
   $ gem list |grep rails
   rails(2.3.2) と表示されることを確認する.

6. OpenSSL 関連のインストール.
   $ sudo apt-get install libopenssl-ruby
   $ sudo apt-get install libssl-dev
   インストール確認をする.
   $ dpkg -l |grep ssl
   ii libopenssl-ruby 4.2
   ii libssl-dev 0.9.8g-10.1ubuntu2.1
   と出力されることを確認する.

7. Sun java 6JDK のインストール.
   途中ライセンス許諾やインストール実行確認があるがカーソルで一番下まで動いて指示に従い tab で y や ok を選択し Enter キーを押して進むこと.
   $ sudo apt-get install sun-java6-jdk
   インストール確認をする.
   $ dpkg -l |grep sun-java
   ii sun-java6-jdk 6-10-0ubuntu2 と出力されることを確認する.

8. Ruby on Rails の動作確認.
   test というプロジェクトの新規作成をする.
   $ rails test
   create ･･･ と複数のフォルダ，ファイルが作成されることを確認する.

9. libxml2-dev および libxslt-dev のインストール.

$ sudo apt-get install libxml2-dev

$ sudo apt-get install libxslt-dev

10. mechanize0.8.5 のインストール.

    $ sudo gem install mechanize

11. nokogiri1.3.2 のインストール.

    $ sudo gem install nokogiri

A.3. web 公開のためのインストールおよび設定

web 公開のためのインストールおよび設定は以下の手順で行われる.

- apache2.2 および関連モジュールのインストールおよび設定ファイルの作成 (設定ファイルの有効化).
- Fasttluead1.0.1 のインストール.
- Passenger 関連のインストールおよび apache 設定ファイルの作成.
- Apache2.2 の VirtualHost 設定.
- Hosts 設定の変更.
- Eruby-1.0.5 のインストール.
- modruby-1.3.0 のインストール.
- apache の ruby 関連設定ファイルの新規作成.

以下，インストールおよび設定を実施する.

```
LoadModule passenger_module /usr/lib/ruby/gems/1.8/
gems/passenger-2.0.6/ext/apache2/mod_passenger.so
#mod_passenger.so まで改行せずに一行で表記のこと
PassengerRoot /usr/lib/ruby/gems/1.8/gems/passenger-2.0.6/
PassengerRuby /usr/bin/ruby1.8
AddHandler cgi-script .cgi
RailsEnv development
```

図 A.1 Passenger 設定ファイル

1. apache2.2 および関連モジュールのインストール.
   apache 関連モジュールをインストールした後，mod_rewrite を有効にする (途中，y/n?と表示されるので，y を入力する).
   $ sudo apt-get install apache2
   $ sudo apt-get install apache2-prefork-dev
   $ sudo apt-get install libapr1-dev apache2-dev
   $ sudo a2enmod rewrite

2. fastthread1.0.1 のインストール.
   $ sudo gem install fastthread

3. Passenger2.0.6 のインストールおよび acpache モジュールのインストール.
   $ sudo gem install passenger
   $ sudo passenger-install-apache2-module

4. apache2 への passenger 設定ファイル作成.
   passenger.load という設定ファイルを新規作成し，vim で内容を編集する.
   $ sudo vim /etc/apache2/mods-available/passenger.load
   編集内容を図 A.1 に示す.

```
<VirtualHost *:80>
    ServerName vega.kmd.keio.ac.jp
    #vega.kmd.keio.ac.jp はユーザの環境ごとに書き直すこと
    DocumentRoot /var/www/alex/public

    <Directory /public>
      AllowOverride Options
      Options +ExecCGI
    </Directory>
</VirtualHost>
```

図 A.2 Virtual Host 設定ファイル

モジュール設定を有効化し，apache2 を再起動する．

$ cd /etc/apache2/mods-available

$ sudo a2enmod passenger

$ sudo /etc/init.d/apache2 restart

OK もしくは　done と表示されることを確認する．

Fail と表示された場合は，vim で編集した内容が間違っている可能性が高いため，スペルミス等がないかどうかを見直す．

5. Virtual Host 設定．

apache の設定ファイル alex を新規作成し，設定内容を編集する．

$ sudo vim /etc/apache2/sites-available/alex

編集内容を図 A.2 に示す．

設定の有効化の後，apache の再起動．

$ cd /etc/apache2/sites-available

$ sudo a2ensite alex

$ sudo /etc/init.d/apache2 restart

OK もしくは　done と表示されることを確認する．

```
      127.0.0.1 localhost
      131.113.136.31  vega.kmd.keio.ac.jp
#この vega.kmd.keio.ac.jp のみをユーザ環境に合わせ書換える.
```

図 A.3 Hosts 設定ファイル

Fail と表示された場合は，vim で編集した内容が間違っている可能性が高いため，スペルミス等がないかどうかを見直す.

6. default サイト設定の無効化と index.html の削除.
   $ sudo a2dissite default
   $ sudo rm /var/www/index .html
   $ sudo /etc/init.d/apache2 restart
   OK もしくは　done と表示されることを確認する.
   Fail と表示された場合は，もう一度 sudo a2dissite default からやり直す.

7. hosts 設定の変更.
   $ sudo vim /etc/hosts
   編集内容を図 A.3 に示す.
   設定の確認をする.
   別の端末で http://vega.kmd.keio.ac.jp/（ユーザ環境に合わせて読み換えること）にアクセスして rails アプリのデフォルトページが表示されることを確認する.

8. eruby-1.0.5 のインストール.
   $ cd /home/USER/sources
   $ wget http://modruby.net/archive/eruby-1.0.5.tar.gz
   $ tar xvzf eruby-1.0.5.tar.gz
   $ cd eruby-1.0.5
   $ vim ./configure.rb

```
$XLDFLAGS = CONFIG["XLDFLAGS"]  を
$XLDFLAGS = CONFIG["XLDFLAGS"] || ""  に修正.
(つまり || "" を書き加える.)
```

図 A.4 eruby 設定ファイル

編集内容を図 A.4 に示す.

$ ./configure.rb --enable-shared --with-charset=euc-jp --prefix=/usr

$ make

$ sudo make install

9. mod_ruby-1.3.0 のインストール.

   $ cd /home/USER/sources

   $ wget http://modruby.net/archive/mod_ruby-1.3.0.tar.gz

   (mod と ruby-1.3.0 の間にアンダーバーがあるので注意.)

   $ tar xvzf mod_ruby-1.3.0.tar.gz (mod と ruby-1.3.0 の間にアンダーバーがあるので注意.)

   $ cd mod_ruby-1.3.0 (mod と ruby-1.3.0 の間にアンダーバーがあるので注意.)

   $ ./configure.rb --enable-shared --with-apxs=/usr/bin/apxs2 --enable-eruby

   $ make

   $ sudo make install

10. apache の ruby 関連設定ファイルの新規作成.

    $ sudo vim /etc/apache2/mods-available/ruby.load

    編集内容を図 A.5 に示す.

    設定ファイルのあるディレクトリに移動，設定ファイルの有効化，apache2 の再起動をする.

    $ cd /etc/apache2/mods-available

$ sudo a2enmod ruby
$ sudo /etc/init.d/apache2 restart
OK もしくは　done と表示されることを確認する．
Fail と表示された場合は，vim で編集した内容が間違っている可能性が高いため，スペルミス等がないかどうかを見直す．

A.4. HyperEstraier 関連のインストール

HyperEstraier のインストールは以下の手順で行われる．

- libiconv1.12：文字コード変換のインストール．
- zlib1.2.3：可逆データ圧縮（バージョン 1.2.1 以降）のインストール．
- QDBM1.8.77：組み込み用データベース（バージョン 1.8.75 以降）のインストール．
- HyperEstraier1.4.13：サーチエンジンのインストール．
- rubynative：ruby バインディングのインストール．

必ず以下の順番にインストールする必要がある．順番通りに行わない場合，途中でエラーが表示されるので最初からインストールをやり直す．以下，インストールを実施する．

1. libiconv1.12 のインストール．
   $ cd /home/USER/sources (USER は各自ユーザ名を入力のこと．)
   $ wget http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.12.tar.gz
   $ tar xvzf libiconv-1.12.tar.gz
   $ cd libiconv-1.12
   $ ./configure
   $ make

$ make check
ここでエラーが出てないかをチェックする．出ている場合は，もう一度 wget からやり直す．
$ sudo make install

2. zlib2.3 のインストール．
   $ cd /home/USER/sources (USER は各自ユーザ名を入力のこと．)
   $ wget http://www.zlib.net/zlib-1.2.3.tar.gz
   $ tar xvzf zlib-1.2.3.tar.gz
   $ cd zlib-1.2.3
   $ ./configure -s(必ず-s をつけること．これがないと HyperEstraier がうまくインストールできない．)
   $ make
   $ make check
   ここでエラーが出てないかをチェックする．出ている場合は，もう一度 wget からやり直す．
   $ sudo make install

3. QDBM1.8.77 インストール．
   $ cd /home/USER/sources (USER は各自のユーザ名を合わせて入力のこと．)
   $ wget http://qdbm.sourceforge.net/qdbm-1.8.77.tar.gz
   $ tar xvzf qdbm-1.8.77.tar.gz
   $ cd qdbm-1.8.77
   $ ./configure - -enable-zlib - -enable-pthread
   （必ずオプションの- -enable-zlib - -enable-pthread をつけること．）
   $ make
   $ make check
   ここでエラーが出てないかをチェックする．出ている場合は，もう一度 wget

からやり直す.

$ sudo make install

4. HyperEstraier1.4.13 インストール.

   $ cd /home/USER/sources (USER は各自ユーザ名を入力のこと．)

   $ wget http://hyperestraier.sourceforge.net/hyperestraier-1.4.13.tar.gz

   $ tar xvzf hyperestraier-1.4.13.tar.gz

   $ cd hyperestraier-1.4.13

   $ ./configure

   $ make

   $ make check

   ここでエラーが出てないかをチェックする. 出ている場合は，もう一度 wget からやり直す.

   $ sudo make install

5. rubynative(ruby バインディング) のインストールとパス指定.

   $ cd /home/USER/sources/hyperestraier-1.4.13/rubynative/ (USER は各自ユーザ名を入力のこと．)

   $ ./configure

   $ make

   $ make check

   ここでエラーが出てないかをチェックする出ている場合は，もう一度 wget からやり直す.

   $ sudo make install

   $ sudo vim /etc/ld.so.conf.d/ruby-estraier.rb

   編集内容を図 A.6 に示す.

   $ sudo /sbin/ldconfig

6. HyperEstraier の検索インデックスの作成.

検索対象にするディレクトリを決め，その上位ディレクトリに移動.
以下は/home というディレクトリを検索対象にした場合の例.
$ cd /
$ sudo estcmd gather -il ja -sd casket /home
とすると，/ に casket ディレクトリが作成されるこれが index のディレクトリである.

A.5. Alexandria Digital Library コンポーネントのインストール

Alexandria Digital Library コンポーネントのインストールは以下の手順で行われる.

- データベースおよびデータベーステーブルの作成.
- Alexandria Digital Library コンポーネントのダウンロードおよび設置.
- 表示確認.

以下，インストールを実施する.

1. データベースおよびデータベーステーブルの作成.
   $ mysql -u root -p
   パスワードはインストール時に入力した各自のパスワードを入力する.
   mysql: create database alex;
   alex に移動してデータベーステーブルを作成する.
   mysql: use alex charset utf8;
   mysql: create table selfInfo(id int not null auto_increment, supernode int null, name text null, port int null, primary key (id));
   mysql: create table notloop(id int not null auto_increment, random int null, primary key (id));

mysql: create table neighborlist(id int not null auto_increment, name text null, port int null, point int null, primary key (id));
mysql: create table nodemapdb(id int not null auto_increment, parentName text null, parentPort int null, hop int null,childName text null, childPort int null, primary key (id));
mysql: create table superlist(id int not null auto_increment, name text null, port int null, primary key (id));
作ったテーブルを確認する.
mysql: describe alex;
mysql からログアウトする.
mysql:exit

2. Alexandria Digital Library コンポーネントのダウンロードおよび設置.
   http://131.113.136.4/dav/svn/alex から Alexandria Digital Library コンポーネントをダウンロードし，/var/www に設置する.
   $ cd /home/USER (USER は各自のユーザ名を合わせて入力のこと．)
   $ svn co http://131.113.136.4/dav/svn/alex/trunk
   $ sudo mv /home/USER/alex/trunk /var/www/alex (USER は各自ユーザ名を入力．)

3. 表示確認.
   別の端末から ブラウザで http://vega.kmd.keio.ac.jp（各自の host 名）/top にアクセスする.

```
LoadModule ruby_module /usr/lib/apache2/modules/mod_ruby.so
# ClearModuleList
# AddModule mod_ruby.c
<IfModule mod_ruby.c>
RubyRequire apache/ruby-run
# /ruby 以下のファイルを Ruby スクリプトとして実行する.
<Location /ruby>
SetHandler ruby-object
RubyHandler Apache::RubyRun.instance
</Location>
# *.rb を Ruby スクリプトとして実行する.
<Files *.rb>
SetHandler ruby-object
RubyHandler Apache::RubyRun.instance
</Files>
</IfModule>
<IfModule mod_ruby.c>
RubyRequire apache/eruby-run
# /eruby 以下のファイルを eRuby ファイルとして扱う.
<Location /eruby>
SetHandler ruby-object
RubyHandler Apache::ERubyRun.instance
</Location>
# *.erb を eRuby ファイルとして扱う.
<Files *.erb>
SetHandler ruby-object
RubyHandler Apache::ERubyRun.instance
</Files>
</IfModule>
```

図 A.5 apache の ruby 関連設定ファイル

```
/usr/local/lib （この一行だけを記入する.）
```

図 A.6 ruby estraier の設定ファイル